

Control Modules - When and Why?

A three-step explanation of a pioneering concept

BACKGROUND

When structured and object-oriented programming began to infiltrate computer science in the 1980s, researchers at Lund Institute of Technology in Sweden began to investigate means to apply this principle to programming of industrial automation systems. Their efforts resulted in the control module concept, which was incorporated in the distributed control system (DCS) *SattLine*, presented at the Interkama Fair in Germany in 1989. ABB's engineering tool *Control Builder*, a part of the *Industrial IT* automation platform, supports an updated and refined version of this concept in the *Professional* version.

INTRODUCTION

Programming with the languages Instruction List, Ladder Diagram, and Structured Text can be regarded as one school of application design, function blocks another, and control modules a third, one that expands the IEC 61131-3 standard and augments function encapsulation.

The object-based principle of control-module programming requires a different way of thinking, and the design phase tends to be more time-consuming than in traditional programming. On the other hand, with control modules it is easy to wrap up – and re-use – all functionality (complex as well as simple AND/OR logic), something that may save plenty of time, shorten testing phases, and yield solutions safer, more flexible, and easier to maintain. Consequently, the use of control modules mostly pays off radically over time. The key is the **code sorting** feature, which sees to that the code executes with respect to data flow rather than program flow.

For the uninitiated, code sorting and its benefits may be a bit hard to grasp. The examples and reasoning presented below will hopefully shed some light over the mechanisms and gains.

WHY USE CONTROL MODULES?

Example 1

Consider a simple plant that allows us to produce an amount of material by requesting a portion of it from a termination tank. A traditional program solution with global variables may look like the one in Fig. 1:

| | |
|--|---------------------|
| Request := Need - Delivered; (*Need is the total amount of material needed*) | Code Block 1 |
| If Request > 0 and Storage >= Request then (*Storage is the total amount of material in the tank. Set initially*) Quantity := 1; (*Quantity is the amount of material being delivered per scan*) Else Quantity := 0; End_if; Storage := Storage - Quantity; | Code Block 2 |
| Delivered := Delivered + Quantity; | Code Block 3 |

Fig. 1 The code of our basic example.

We set the *Need* variable (to, e.g., 50) and the program starts. Quite straightforward, in other words and indeed, if we work with simple applications that we don't have to modify later on, a traditional solution with a straight program flow is the easiest way. For larger programs, a function block solution yields a better overview, but in this particular application we have a bi-directional data flow, and, therefore, it is not all that suitable to implement with function blocks (Fig. 2).

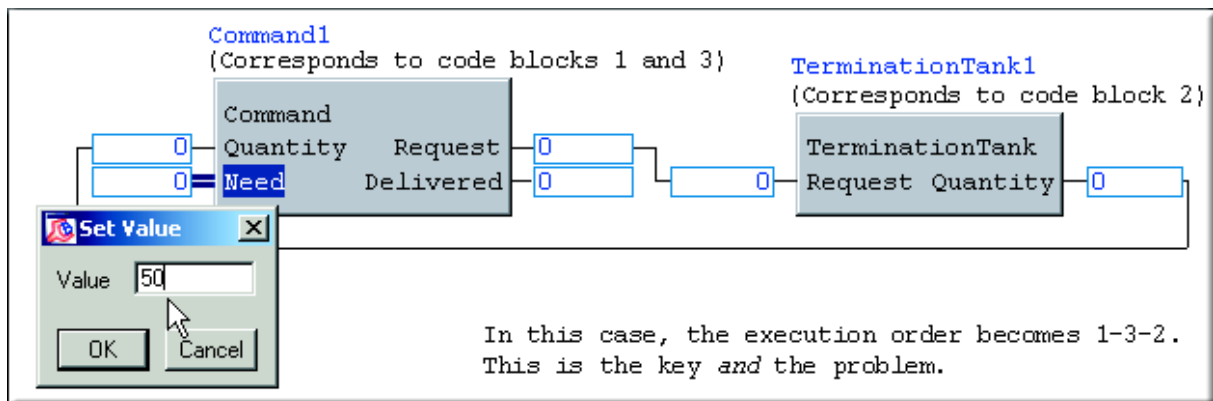


Fig. 2 Our basic design represented by a function block diagram (screen capture from the function block diagram editor of *control builder* at design time).

If we do, time delays arise (Fig. 3), even when the application runs in the same task in one controller, unless we separate the code into several function blocks that we sort and connect manually. Time delays won't be a problem if we use control modules because all code executes in another order, and the *Quantity* variable is not updated to the *Command1* block until the next scan. However, a control module solution requires more effort at the initial design stage, as described below.

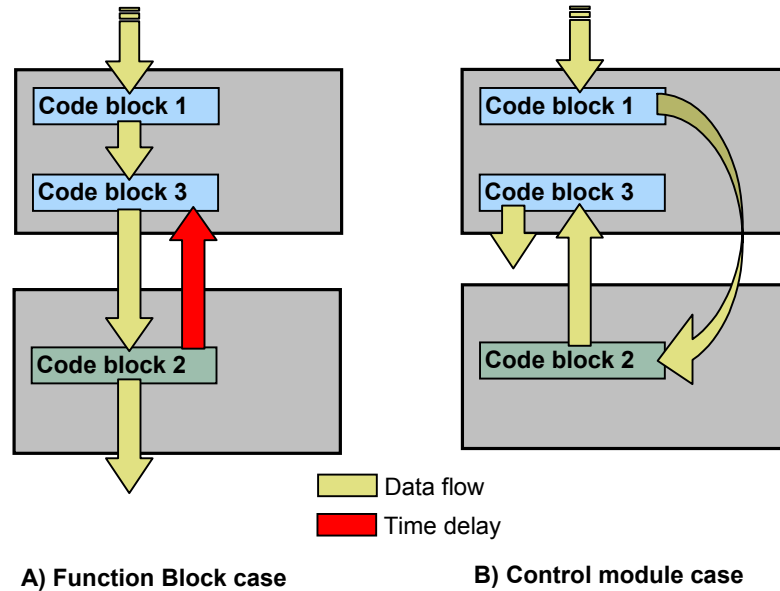


Fig. 3 Comparison between the respective orders of code-block execution.

In the control-module based solution, we implement the application by writing the code blocks and declaring required variables and parameters, drawing the two objects (the command module and the tank module) and connecting the parameters graphically (Fig. 4). Here, we use structured parameters for handling the data flow between the control module instances.

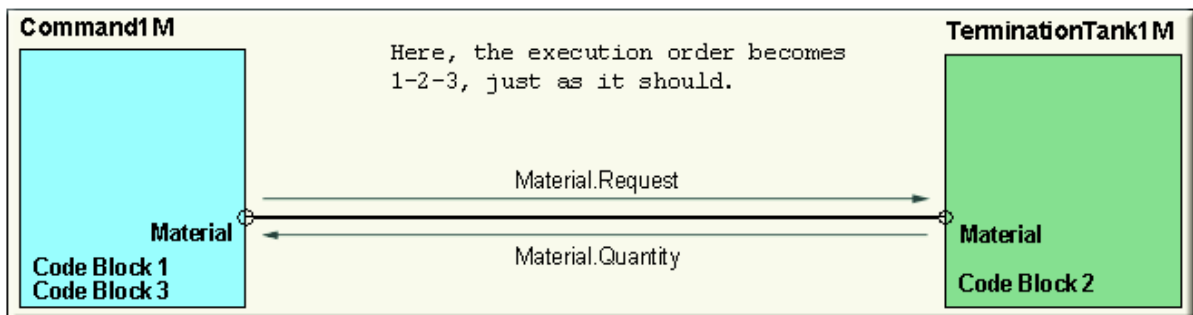


Fig. 4 Our application with one termination-tank control module. The arrows denote value-exchange directions (screen capture from the Control Module Diagram editor of *Control Builder* at design time).

We will now make use of the code sorting, which can be described as that the compiler analyzes the code with respect to data flow – not program flow – and determines the optimal code block execution order. In turn, all reading of and writing to variables in an application designed with control modules take place during a single execution scan, and all data exchange between the control-module instances occur in correct order.

To make code sorting possible, we separate the code in the command module, *CommandIM*, into two code blocks (corresponding to code blocks 1 and 3 in the “traditional” example, (see Fig. 1) one that handles the query (the request) to *TerminationTankIM*, and one that handles the answer. The code in Fig. 1 can be used here as well, with minor modifications only. (In a way, control modules can be regarded as another means to organize the code.) Later on, when we download the application to a controller, the compiler will determine the optimal code-block execution order. This is the essence of code sorting and control modules.

The input to *CommandIM* is the needed amount of material, i.e., the total amount to be requested. The request is passed to the tank via a graphical parameter connection and the material is dispensed in small amounts per scan. The application stops when the requested amount is delivered.

Now suppose we want to expand our plant to include an intermediate tank; with a traditional solution, we face a rather complex situation where we need to declare a number of new variables and work out which variables are to be written and read, respectively. In other words, keep thorough track of what is happening in our code. Fig. 5 shows the result of this operation, which requires ample rewriting of the code given in Fig. 1.

```

Request := Need - Delivered; Code Block 1

IntermediateQuantity := 0;
TerminationRequest := 0;
If Request > 0 then
  If IntermediateStorage >= Request then Code Block 4
    (*Storage is the total amount of material in the tank*)
    IntermediateQuantity := 5;
  Else
    TerminationRequest := Request - IntermediateStorage;
    (*Reducing Setting the request equal to the intermediate tank deficit*)
  End_if;
End_if;

TerminationQuantity := 0;
If TerminationRequest > 0 and TerminationStorage >= TerminationRequest then
  TerminationQuantity := 5; Code Block 2
End_if;
TerminationStorage := TerminationStorage - TerminationQuantity;
(*Quantity is the amount of material being delivered per scan*)

IntermediateStorage := IntermediateStorage - IntermediateQuantity;
Quantity := IntermediateQuantity + TerminationQuantity; Code Block 5

Delivered := Delivered + Deliver; Code Block 3

```

Fig. 5 Our new, extended code, which includes the function of the intermediate tank as well.

Despite the significant changes, we would probably be able to manage without too much workload in this simplified example application, but things will get tricky if we need yet another tank later on. In fact, for each additional tank, the situation becomes more difficult to sort out. Likewise, even with only three blocks, the function block diagram begins to look a little messy, and time delays are unavoidable (Fig. 6). A design with five function blocks would take care of the time delays, but look rather chaotic. (Here, we have followed the custom of using simple data types.)

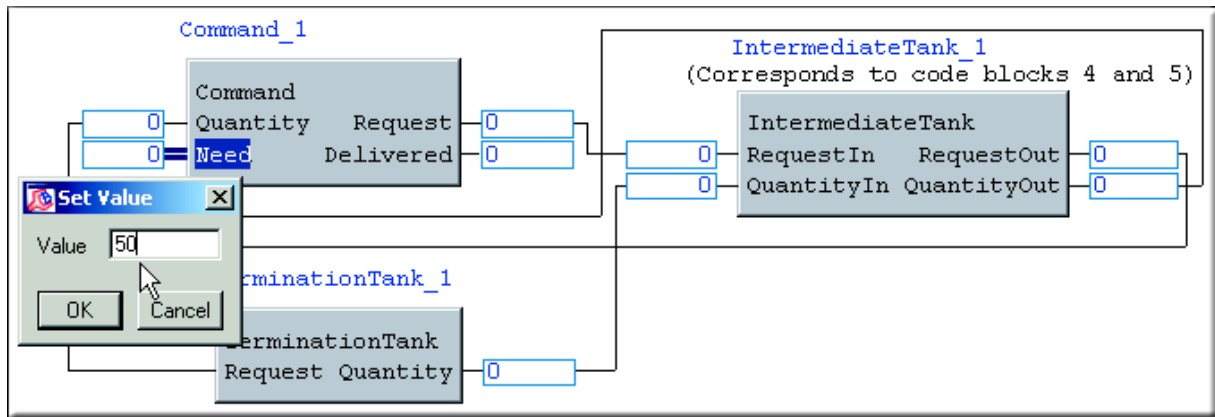


Fig. 6 Our extended design represented by a function block diagram.

On the other hand, if we started out with a control module solution, we can design an intermediate tank, insert it between *Command1M* and *TerminationTank1M* and declare the new parameters and any new variables. Next, we connect the new parameters graphically (Fig. 7) and add the code to be executed by the new tank control module (*IntermediateTank1M*, code blocks 4 and 5); the code sorting relieves the programmer of the time-consuming and error prone procedure of ensuring that all variables are written and read in the correct order.

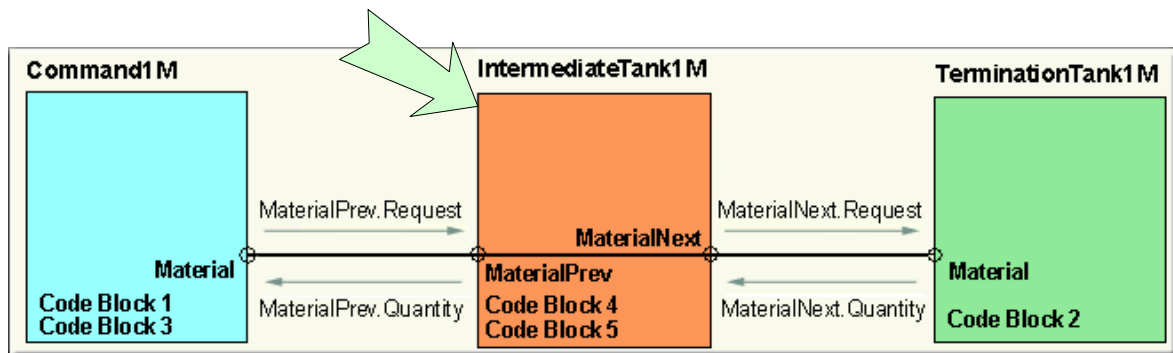


Fig. 7 Our application completed with another tank control module instance.

Fig. 8 illustrates the code block execution order for this new case. There's now two time delays in the function block case.

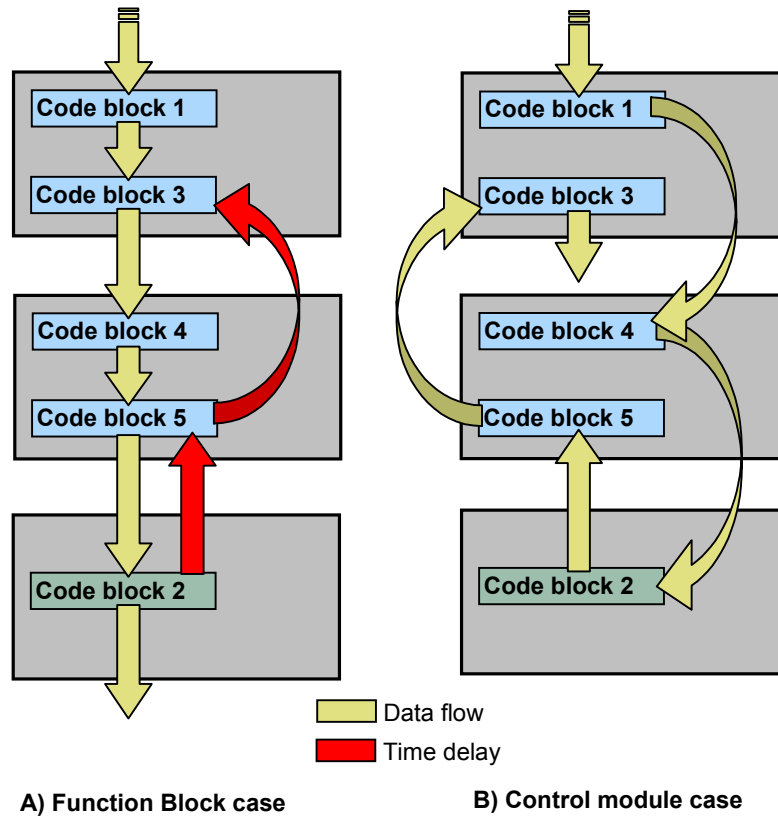


Fig. 8 Data flow and execution order for the case with an intermediate tank.

The design of the new tank does require some thinking, but once implemented we can re-use it simply by inserting as many additional tanks as we like - the code sorting routine will see to that all variables are read/written in the correct order. Concisely, the larger the number of similar objects we insert, the more we gain from using control modules, and the more work we face when using function blocks. Moreover, the time delays may become unacceptable when we use function blocks for a large number of tanks, and the possibility to monitor the program diminishes considerably.

In summary, the more complex a process application, and the more flexibility (e.g., modifications, expansions) required, the more rational to use control modules. Likewise, when the final design isn't entirely determined beforehand, trial-and-error analyses with control modules is typically a constructive approach. Apart from the automatic code sorting, the extensive re-use possibilities and the free-graphics editor constitute the real benefits of control modules. Below follows an example.

Example 2

In example 1, we saw how an application structure could be expanded along its top level. Here, we will see that the control module concept and the code sorting mechanisms apply equally well to hierarchical structuring (that is, control modules within control modules) in any number of levels. In other words, we can rather easily combine basic types with more specialized ones to form new types that suit our needs. Consider two existing control module types, one that represents a reactor and one that represents a pump (Fig. 9). The reactor type contains one sub-control module instance each of a valve and a motor running a mixer. Now say that we need another reactor type, one that includes the pump as well. We create a new, empty type (1) and insert one instance each of the reactor type (2) and the pump type (3). One day, the need to more accurately control a flow somewhere else in our plant arises and we need a reactor type with one more pump, a smaller one. We copy our new reactor type, rename it (4) and insert a new instance of the pump type (5). (The free graphical layout permits us to draw the lower-capacity pump a bit smaller than the other for clarity).

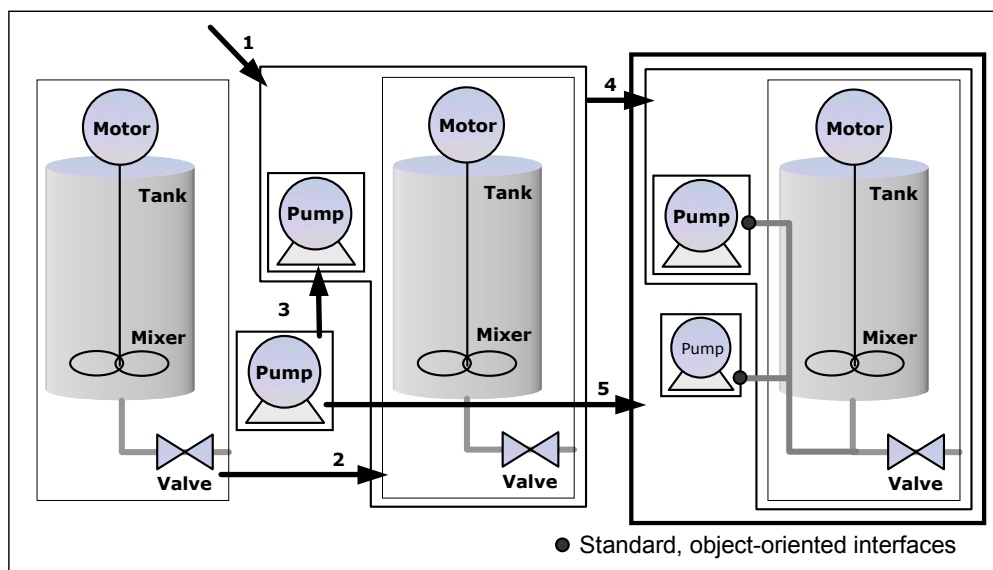


Fig. 9 Re-use working principle.

We can now reflect the capacity difference simply by modifying a parameter (provided, of course, that the initial pump type has been properly designed).

Apart from some coding and parameter connections we are done; at compile time, the code sorting routine will again take care of the data execution order. It is important to emphasize that this hierarchical re-use of process objects and the associated, limited code changes are unique to control modules; a plain program- or function block approach would require considerably more effort.

In addition, the free graphics also permits us to show only important process features and hide details such as AND/OR gates or other glue logic, and thereby obtain a better program status overview. Yet another advantage is that the overall plant design becomes more straightforward - the graphical configuration can be drawn more or less directly from the flow chart.

However, control modules can be used to encapsulate and represent sheer functions as well, not only physical process objects. Typical examples are standardized control logic functions (e.g., interlocks, error handling, and process-object interfaces). The subsequent re-use of these simplifies the application design process significantly and saves plenty of time and money. In example 3, we look further into this principle and its advantages.

Example 3

Here, we use control modules to design a realistic application involving silos, bins, hoppers, and conveyers (Fig. 10). As we go, we will clearly see the benefits of the control module concept.

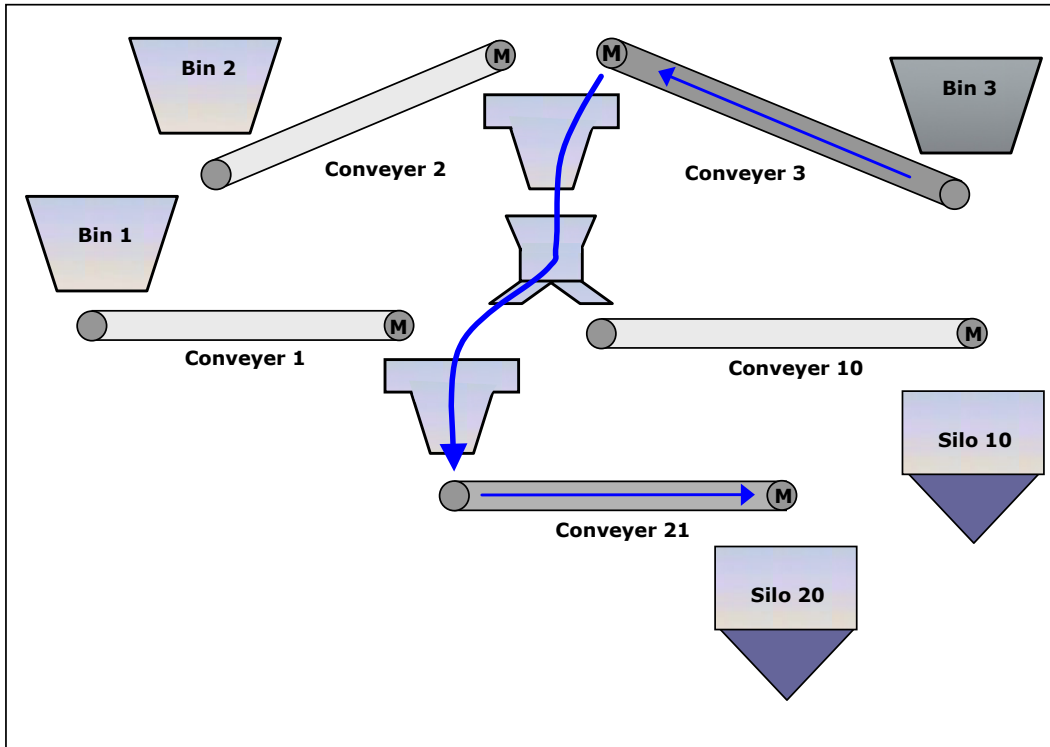


Fig. 10 A schematized view of our initial material transport application.

All conveyers in Fig. 10 are instances of the same control module type. They contain sub control-module instances that hold basic logic for control of belt speed, alarms, communication interfaces with neighbors, and so on.

In this context, it is suitable to emphasize the intended use of function blocks and control modules, respectively. A function block should be used to represent, e.g., a simple motor, of which we make variants by modifying the surrounding glue logic. A control module is suitable for representing a complex motor (including all interlocks, glue logic, etc). Subsequently, we implement any variants by altering parameter connections.

Now, if we, for example, request that Silo 20 should receive material from Bin 3 via conveyers 3 and 21, a status query is passed to all objects along this transportation route (marked by arrows). When the query returns a go-ahead signal from each conveyer's communication interface, the operation can commence. With examples 1 and 2 in mind, we may conclude that once one conveyor control module type (including its sub control-module types) is implemented, it can be instantiated and re-used merely by changing its identity and I/O- and neighbor connection parameters. Therefore, we may quite simply insert another set of hopper and conveyor instances in the structure, (Fig. 11) rename them, and modify their connections with the surroundings.

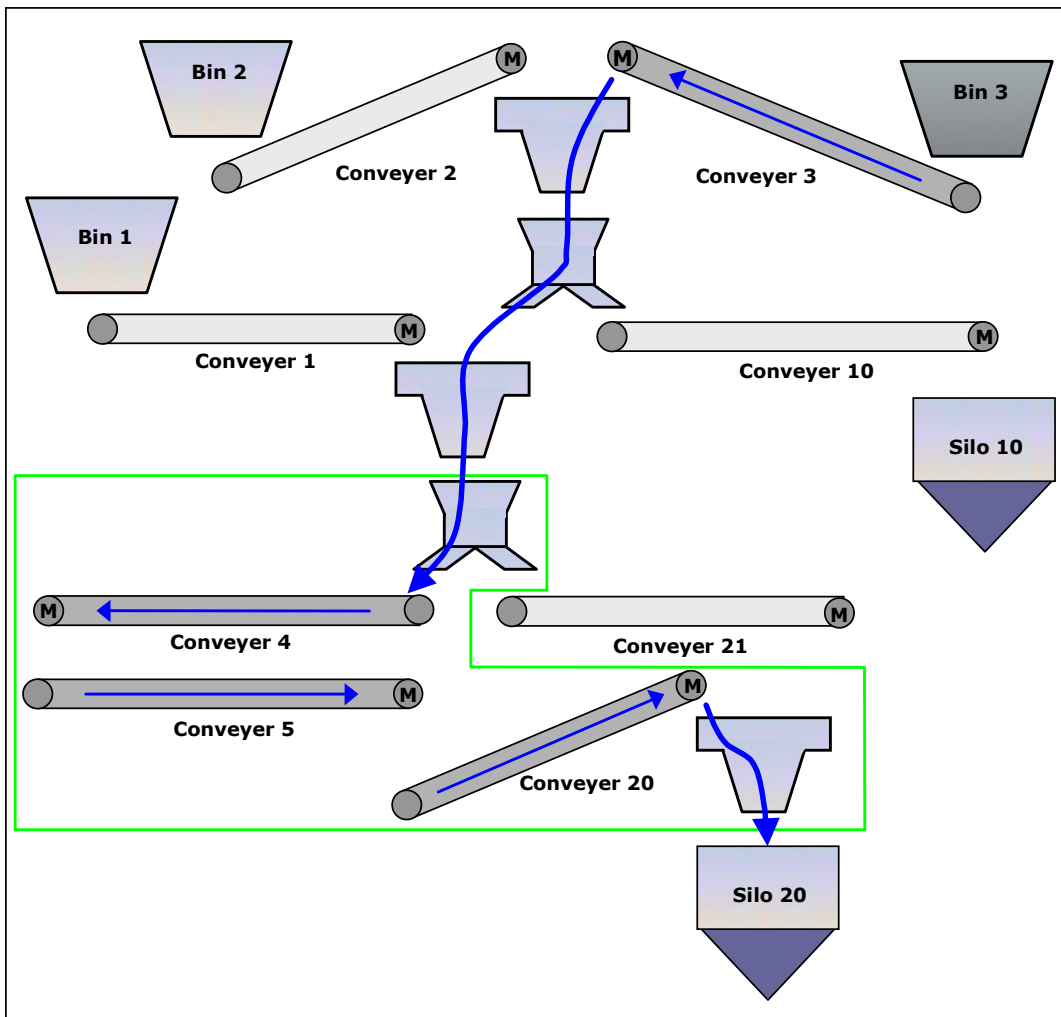


Fig. 11 A schematized view of our material transport application extended with three conveyers and two hoppers. The added objects are located within the green frame.

If we now issue another request, stating that Silo 20 should receive material from Bin 3 via conveyers 3, 4, 5, and 20 instead, we receive answers from the newly inserted and connected objects as well. Everything seems OK, and the material transport starts along the new route. Thanks to the code sorting, and that we have put some well-contemplated effort into the initial design of the various objects, the modified structure can be up and running in a matter of minutes, with maintained quality. If this structure had been traditionally designed with programs and function blocks, we would likely face a long (perhaps several days) testing/ functionality verification phase to assure the quality. All in all, the key to successful use of the control module concept is thorough knowledge and consideration of the problem at hand. Then, the benefits of application programming with control modules will not be long in coming!

FAQ

Q: What kinds of applications are most suitable to implement with control modules?

A: *Dairy- and pulp-and-paper applications, for example. The many complex object dependencies make a solution with control modules the most rational choice.*

Q: What kinds of applications are not suitable to implement with control modules?

A: *Control applications with straightforward execution and low level of re-use. In this case, conventional Function Block or Ladder programming yield simpler and cheaper solutions.*

Q: What kinds of functions/process objects are most suitable to represent with control modules?

A: *This is a tricky one. No strict rules apply, but use of control modules below a certain complexity level, such as AND/OR logic and simple valve logic, is normally not justifiable. If subsequent object re-use is anticipated, control modules are the better choice. If a given function can be expressed in a single code block, it is typically easier to design with a function block.*

Q: Is it possible to put function block instances within a control module type?

A: *Yes. As a matter of fact, this is the normal way of designing control modules. The function blocks are here used for small building blocks with simple parameters, e.g., timers and alarm detection.*

Q: Can I handle I/O-connections from within control modules?

A: *Yes. In object-based programming, it is natural to encapsulate the I/O-functionality within the object and control its behavior via parameters. This principle goes for function blocks as well.*

Q: How does the code sorting work?

A: *The code sorting is based on that the compiler analyzes how code blocks read or write variables, with the result that a code block that writes to a variable is executed before a code block that reads that variable. A key is the static parameter connections, which do not change during execution, only via code changes and recompilation. The static connections are set at compile time and yield efficient code generation; powerful, compiler-based program analysis; and better performance.*

Q: OK, but what if two code blocks read and write to the same variable?

A: *This situation results in a program loop, which can be resolved by declaring the variable as "state". By doing this, the variable will hold two values, one old and one new, and thereby break the loop.*

Q: Do I have to use graphics when designing with control modules?

A: *No. Again the key thing about control modules is the code sorting. Parameter connections can be established textually as well. Yet, the graphics improves the overview of the application and facilitates design modifications.*