Design of Controllers for Plug-And-Play Composition of Automated Systems from Smart Mechatronic Components

Valeriy Vyatkin* and Hans-Michael Hanisch**

**The University of Auckland,*
New Zealand
v.vyatkin@auckland.ac.nz

***Martin Luther University of Halle-Wittenberg,*
Halle, Germany
hans-michael.hanisch@iw.uni-halle.de

Abstract - **This paper discusses two heuristic approaches to the design of distributed controllers appropriate for plug-and-play integration of mechatronic systems. The first approach is based on the Master-Slave relationship between the mechatronic objects. The second approach relies on a constraint-based protocol, when controllers permit/block the actions of other controllers.**

1 INTRODUCTION

The development of automated systems and machines from the ready-made blocks, known as mechatronic components, aims at rapid engineering and re-engineering of automated manufacturing systems. This, in turn, implies that the mechatronic components have most of their operations "pre-automated" by their vendors. A system integrator (or a machine designer) puts the mechatronic components together and designs the overall control of the system re-using the existing controllers of the mechatronic components.

Then, the resulting control code is deployed either on a single control device (for instance, a Programmable Logic Controller) or on a distributed network of such devices.

The described engineering approach gives rise to a number of questions, like:

- Is there a standard interface or interfaces for "gluing" the controller components into the controller of the system with minimum effort and in a systematic way?
- Can the inheritance concept be applied to the controllers of mechatronic systems, so the controller of an object with more options can be built incrementally extending the controller of a simpler object?
- What are the limitations of execution platforms (e.g. PLCs or distributed embedded devices) for implementing the controllers that are composed using such standard interfaces?

In this paper we present two approaches to decentralized logic control design that do not assume any coordinating master controller of the whole system. The system is supposed to implement a production goal by implicit collaboration of the local controllers of its components.

The first approach relies on the Master-Slave relationship between the mechatronic components and their processes. In this case the controller of the master object gives explicit commands to the controller of the slave. The relationship is determined by the physical properties of the plant and by the roles of each mechatronic component in the process.

The second approach assumes the objects to be of the same tier. Their communication is based on imposing constraints to the behaviour of other parties. The paper is structured straightforward according to these approaches.

2 MASTER-SLAVE CONTROLLERS

2.1 Layered architecture of controllers

Our intention is to investigate the approaches that would allow more flexibility in building systems from mechatronic units like from the blocks with autonomous control. For that we propose application of a layered architecture with three layers as presented in Figure 1.

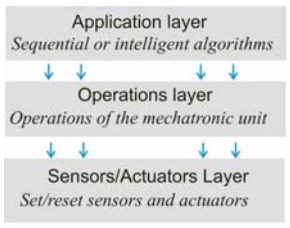
Application layer
Sequential or intelligent algorithms
Operations layer
Operations of the mechatronic unit
Sensors/Actuators Layer
Set/reset sensors and actuators

Figure 1. Layers of the distributed control architecture.

The layers have the following functionality:

1. The a*pplication layer* includes sequential centralized or local decentralized controllers, implementations of agent algorithms, etc.

2. The *operations layer* implements a set of operations defined for mechatronic units along with their implementation sequences.

3. The *sensors/actuators layer* provides direct access to sensors and actuators of the controlled object.

The functions of the higher levels use services of the lower ones. If the operations layer controller is present it can encapsulate also the sensor/actuator layer.

2.2 Illustrative example

The layered design is illustrated by means of the following example. The object shown in Figure 2 consists of two pneumatic cylinders moving back and forth. Each cylinder is controlled by signals **move1** and **move2** correspondingly. Cylinder 2 is mounted on the tip of the cylinder 1 and is directed orthogonally. On the tip of the cylinder 2 there is a platform for a work piece. Cylinder 1 is different from cylinder 2 in that it has an additional sensor of middle position.

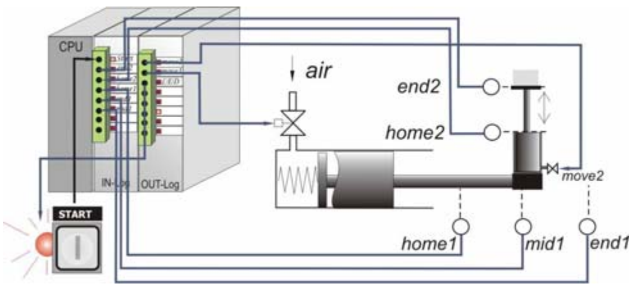

Figure 2. System of two cylinders.

Initially both cylinders are retracted and the LED is lit. That means the system is ready.

The desired behaviour is as follows:

Extend the cylinder 2 to the middle position and then start extending the cylinder 2 simultaneously with the cylinder 1. When both cylinders are extended, the operation of work piece delivery is over. The button's LED goes on and a next push to the button shall return both cylinders to the retracted position.

This behaviour is captured in the time-state diagram shown in Figure 3.

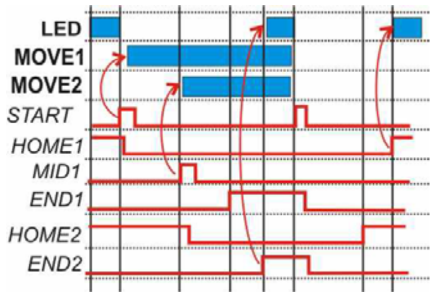

Figure 3. Timing diagram of the
Cylinder system with Master-Slave control.

We are going to design a distributed controller of this system that will consist of three parts as follows:

1. Controller of the button and LED;
2. Controller of cylinder 1 – slave to the button;
3. Controller of cylinder 2 – slave to the cylinder 1;

2.3 Interfaces of a controller in the Master-Slave controller relation

In the remainder of this paper we are assuming that a controller may use (i.e. implement) several interfaces.

In our approach, the most basic interface every controller needs is the interface with the plant. For example, our cylinder controller requires the following plant interface.

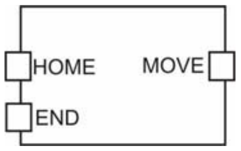

Figure 4. Interface for interaction with plant.

The *plant interface* is determined by the available sensors and actuators of the physical plant. In the graphical notation for interfaces we assume two different kinds of inputs and outputs: data (represented by quadrant boxes) and events (represented by rhombus boxes).

Plant interfaces usually have only data inputs and outputs that correspond to the traditional implementation with Programmable Logic Controllers (PLCs) which are connected to the plant by means of logic level signals.

Besides, a controller may be used as a slave in connection with other controllers or can be a master with respect to other controllers. For supporting this relationship, interfaces are needed for both interactions with the master and with slaves. The interface shown in Figure 5 serves for interaction with a master controller.

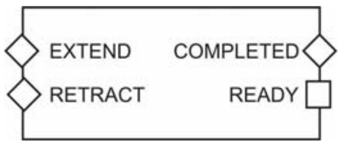

Figure 5. Slave interface for communication with a master controller.

The master and slave interfaces can be less dependent on a particular plant than the plant interface. For example, the interface for interaction with a slave controller can be quite generic as illustrated in Figure 6.

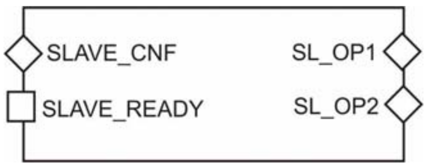

Figure 6. Master interface for interaction with a slave

This interface assumes that the slave can perform two operations, which are invoked by the output events SL_OP1, SL_OP2. The slave reports on either operation's completeness by an event sent to the event input SLAVE_CNF. When the slave is ready for performing either of the operations it sets the SLAVE_READY condition to TRUE.

The layered architecture leads then to the stack of interfaces for each particular controller. This will be illustrated in the next section.

2.4 Bottom-up development of controllers

In this section we introduce a methodology of systematic controller development that follows the bottom-up direction - from simpler controllers to more complex ones. The increase in complexity and functionality is driven by the need to implement more interfaces.

We show, that if controllers of components are designed according to this methodology, they can be combined to the controllers of more complex systems in a plug-and-play way.

We are using the following notation. The controller's semantic is represented by State Charts which is a way to combine the expressiveness and clarity with the formality. State Charts are encapsulated in a box, to which we add the interfaces the controller implements.

2.4.1 Step1: Design the plant level controller

The first step of the process is to design a controller of the lowest level, i.e. the *plant level*. A controller of this level interacts only with sensors and actuators of the plant.

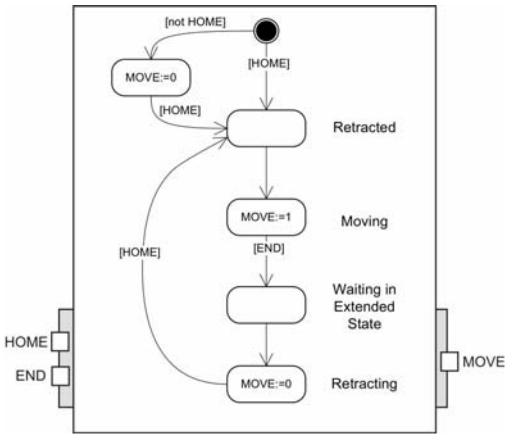

Figure 7. The controller of the cylinder of the plant level.

Such a controller can be obtained by transformation of a State Chart of the desired behaviour of the plant, attaching the corresponding sensor values to the transitions and control signals assignments to the states. An example of the plant level controller of a cylinder is shown in **Figure 7**.

2.4.2 Add implementation of the slave interface (operations layer).

In Figure 8 some new arcs and underlined operators are added to implement the new interface. Thus, both interfaces coexist in the new state chart.

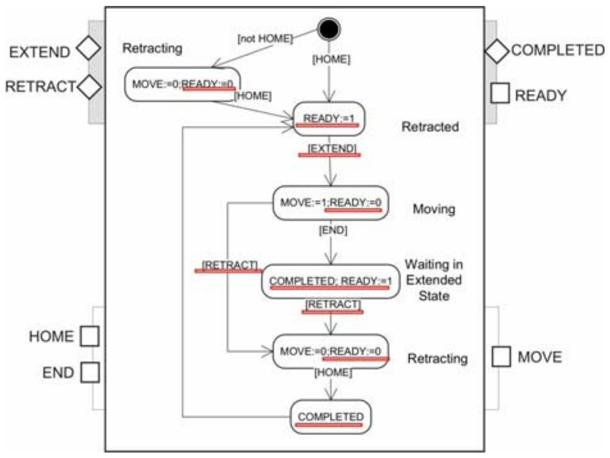

Figure 8. Controller of the cylinder implementing the plant level control and the slave functionality of the command level.

2.4.3 Implement master level interface

If the controller is a master with respect to other controllers, then it has to implement the master interface. The master interface is a mirror to the slave interface.

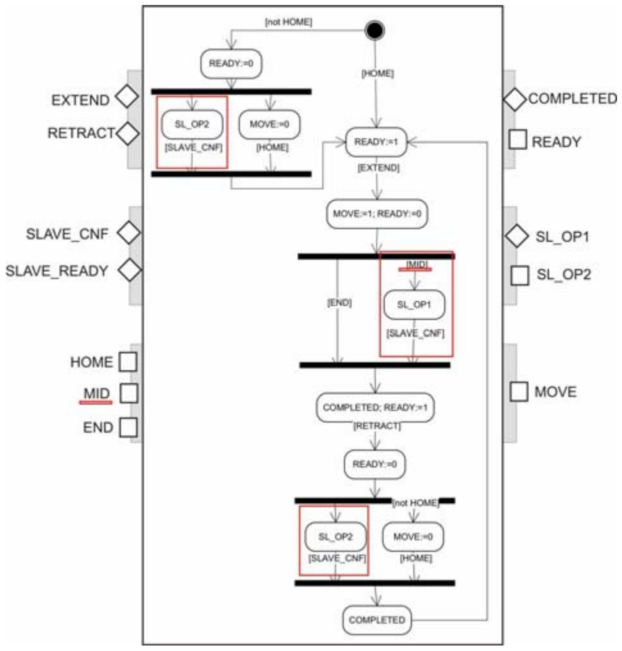

Figure 9. Controller of the cylinder that implements master interface.

The controller of cylinder 1 has additional input MID if compared to the original controller from Figure 7. The sequences needed to implement the master interface are outlined in the boxes and placed in parallel branches.

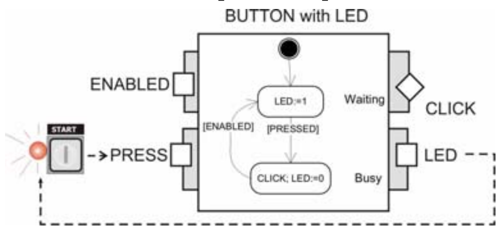

Figure 10. Controller of a button with LED.

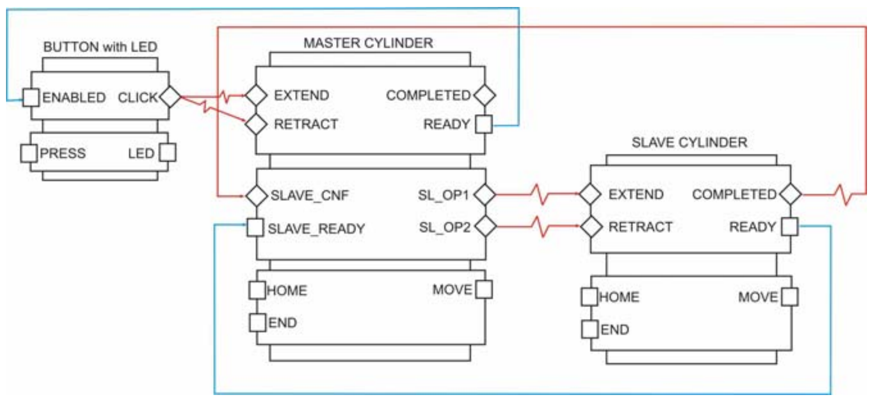

Figure 11. Distributed controller of two cylinders with a button.

Finally, let us consider the implementation of the controller of the button in Figure 10 that implements two interfaces: plant and master. The resultant distributed controller is shown in Figure 11.

3 SAME TIER CONTROLLERS

For illustration we will use a simple model of a manufacturing system called "Distribution Station" (Figure 12) that consists of two *mechatronic units: a feeder unit with a magazine of workpieces and* a pusher; and *a simple manipulator* (*transfer unit*) that takes workpieces from the feeder (left position) and brings them to the opposite position (called next position or right position), where they are supposed to be taken by other automated machines.

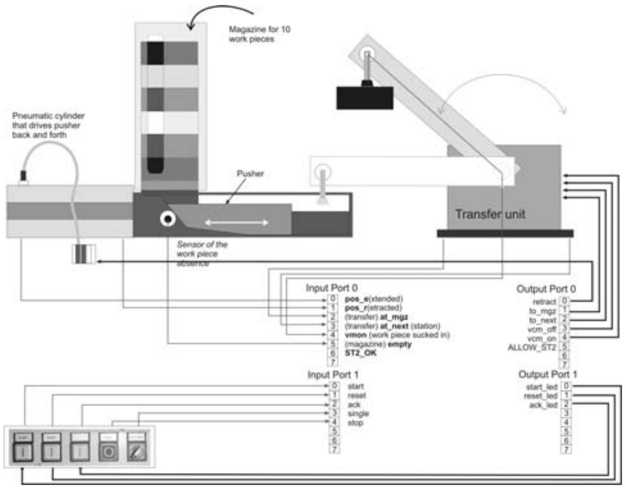

Figure 12. Mechatronic system for work piece storage and transfer [5].

Actually, the "Distribution Station" is a part of a bigger manufacturing system model, which is a chain of several stations representing different stages of a manufacturing process.

In addition to the mechatronic models of machines, the station includes a small panel with buttons RESET, START, STOP, and ACK(nowledge). The buttons have no memory, so each button generates a logical one (TRUE) value as long as it is pressed and zero otherwise. Thus, a short push on a button generates a pulse. The buttons are lit underneath by LEDs that also can be controlled, i.e. whose can be set/reset by the control device as needed. The highlighted buttons may indicate that they are enabled.

The State Chart of a (centralized) controller of the distribution station is shown in Figure 13. The State Chart can be quite trivially converted to the executable code in one of the standard programming languages of Programmable Logic Controllers , such as Ladder Logic or Sequential Function Charts, defined in the IEC 61131-3 Standard [6].

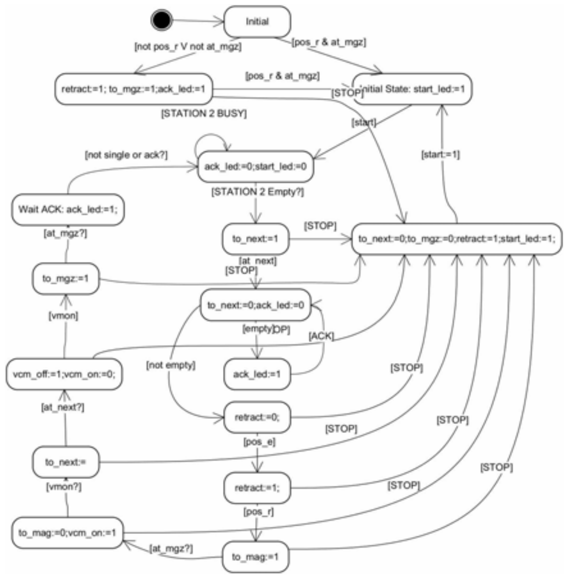

Figure 13. State Chart of the centralized controller [5].

As it is clear from the state activity diagram presented in Figure 14, the process includes well separatable sequences of actions of the feeder and of the transfer units. The processes are partially concurrent.

As applied to the example discussed above, the idea of distribution consists of "splitting" of the centralized control State Chart onto two controllers of the feeder and the transfer respectively, and adding synchronisation of the processes.

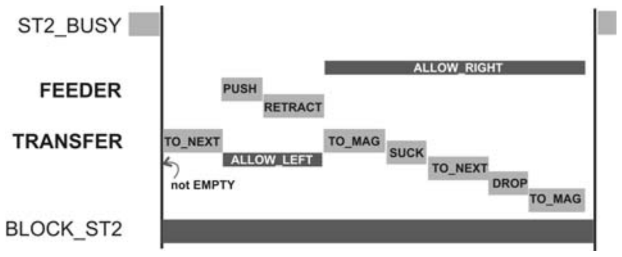

Figure 14. Process activity diagram [5].

For illustrative purposes of this paper we suggest a simple inter-object interface and protocol that is as follows.

Each controller is designed so that it attempts to perform operations as soon as they are not blocked by the controllers of other objects. This implies that the controllers have to be aware of other objects around and of the operations they perform.

The interface is based on the mutually exclusive access to the areas where mechanical parts can clash. Usually these are the areas where the material transfer occurs. In our example such an area is the "End position" of the FEEDER unit where the workpiece is picked up by the TRANSFER unit.

Access to such shared areas can be implemented by standard mutual exclusion algorithms, such as semaphore-based central algorithm, or distributed Ricart and Agrawala algorithm [3]. In this paper, for the sake of simplicity, we are using a simpler mechanism based on passing Boolean permission or blocking conditions from one controller to the other.

The controllers' interface is illustrated in Figure 16. The operation of each mechatronic unit is essentially autonomous, but some actions have guard conditions "allowed by left/right neighbour" (LEFT OK/RIGHT OK). The permission or blocking conditions are set by the controllers of the neighbours on the right/left (if any). This approach assumes a linear order of connections of mechatronic units in the production process. Process systems with arbitrary structure require more sophisticated interaction mechanisms that are under development.

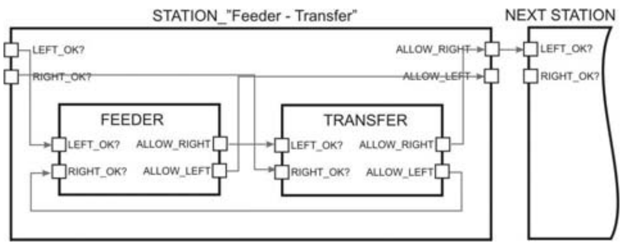

Figure 16. Signal interface of distributed controllers in the permit/block protocol [5].

As seen from Figure 16, this approach also fits well to the hierarchical structure of mechatronic systems. Our sample system has a neighbour on its right side that is "Sorting Station". The stations interface each other exactly same way as their components, i.e. using permissions from their left/right neighbours. As it is seen from Figure 16, the permission from the right neighbour is passed down to the component controller of the transfer station which is physically interacting with the Sorting Station. If the neighbour station is not present, then the permission is set to the constant "TRUE" as it is the case in our example for the left neighbour of the distribution station and the Feeder unit.

The detailed implementation of the Distribution Station controller that consists of FEEDER and TRANSFER controllers communicating via common Boolean variables is presented in form of concurrent state charts

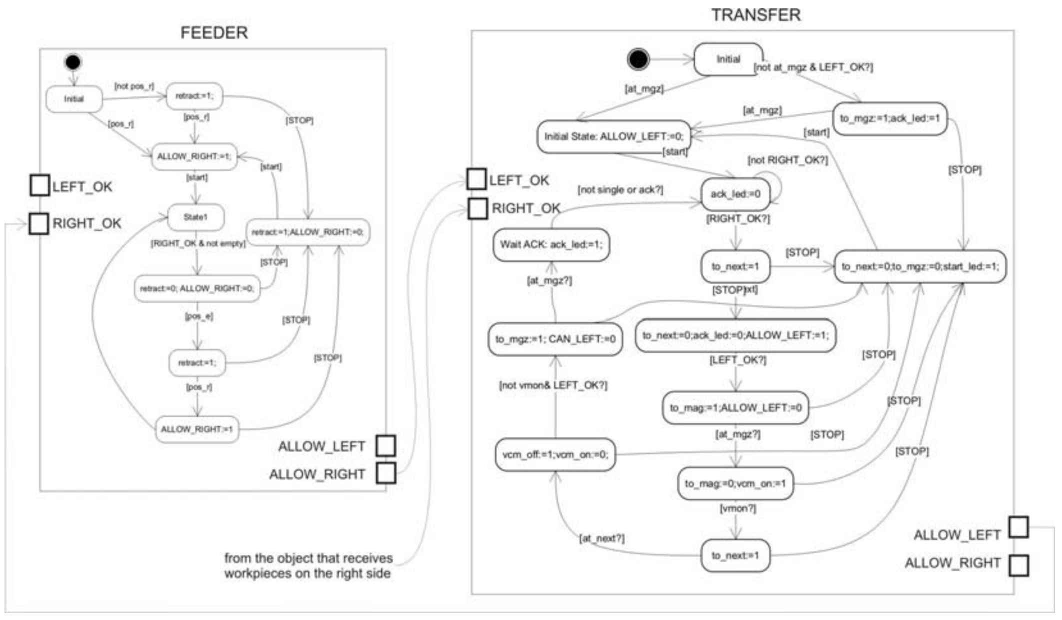

Figure 15. A distributed algorithm that uses the permit/lock protocol via common variables [5].

in Figure 15. Confronting the controllers with the layers description in Figure 1, one sees that the controllers implement the application layer functionality directly interacting with the sensor/actuator layer.

4 IMPLEMENTATION

For demonstration of the concept the decentralized controllers were implemented in IEC 61499 function blocks [2] and simulated in FBDK/FBRT execution environment [4] with minor modifications.

Here we briefly discuss the controller of the Feeder-Transfer system. More complete discussions can be found in [5]

A simplified interconnection of the function block controllers is shown in Figure 17. The inter-controller communication is implemented via passing event and data signals. Thus, event output OUT_CMD of the FEEDER controller is connected to the event input IN_CMD of the TRANSFER controller and vice versa. The data inputs LEFT_OK, RIGHT_OK are associated with event IN_CMD, and the data outputs ALLOW_LEFT, ALLOW_RIGHT are associated with event OUT_CMD. The output event needs to be issued anytime one controller changes the block/permit variables.

The "permit/block" protocol is implemented as follows. Both controller function blocks have (Boolean) inputs LEFT_OK, RIGHT_OK, and outputs ALLOW_LEFT, ALLOW_RIGHT following the pattern presented in Figure 16.

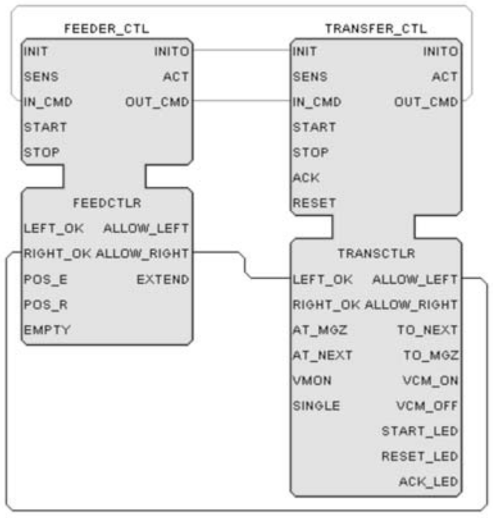

Figure 17. Interacting controllers of FEEDER and TRANSFER encapsulated in function blocks (process interface is omitted) [5].

The ALLOW_RIGHT output of the FEEDER is connected to the LEFT_OK input of the TRANSFER, and the ALLOW_LEFT output of the TRANSFER is connected to RIGHT_OK input of the FEEDER.

The controllers' State Charts are implemented as ECCs of the corresponding function blocks (for example, the Feeder's controller is shown in Figure 18). The variables are set in the algorithms, most of which set/reset just one variable. The algorithms have self-explanatory names, for example the algorithm TO_MGZ1 consists of one operator: TO_MGZ:=*true*; and TO_MGZ0 of: TO_MGZ:=*false*;

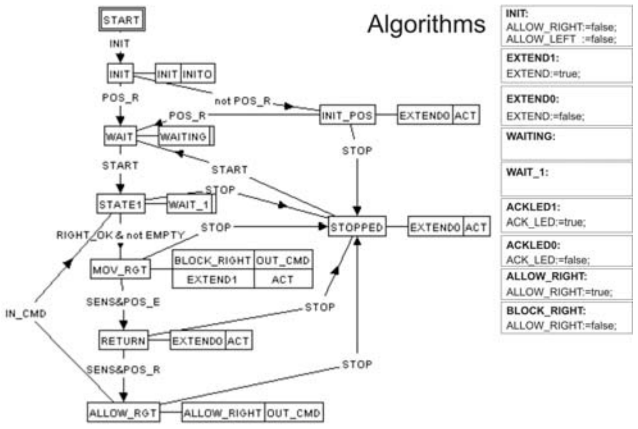

Figure 18. Controller of the Feeder in the ECC form [5].

5 CONCLUSION

In this paper we presented a heuristic methodology for designing autonomous controllers of mechatronic objects. Once the corresponding objects are put into a production system, the system could start its operation without any master controller coordinator.

Certainly this approach is of limited applicability in industry in its pure form. However, we believe that some of its elements can be effectively used as building blocks of autonomous self-configured controllers.

6 REFERENCES

1. IEC61131 - International Standard IEC 1131-3, Programmable Controllers - Part 3, International Electrotechnical Commission, 1993, Geneva, Switzerland
2. IEC61499 - Function blocks for industrial-process measurement and control systems - Part 1: Architecture, International Electrotechnical Commission, Geneva, 2005
3. G. Couloris, J. Dollimore, T. Kindberg, Distributed System: Concept and Design, Addison-Wesley, 2005
4. Function block development kit (FBDK) – Online - www.holobloc.com
5. Vyatkin V., IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design, 271 p., Instrumentation Society of America, 2006