Accepted for IEEE International Symposium on Object Oriented Realtime Computing (ISORC 2001)

Integrating UML-RealTime and IEC 61131-3 with Function Block Adapters

Torsten Heverhagen, Rudolf Tracht
University of Essen, Germany, FB 12, Industrial Automation
Torsten.Heverhagen@uni-essen.de, Rudolf.Tracht@uni-essen.de

Abstract

In this paper we introduce a new UML stereotype, the Function Block Adapter (FBA), which is responsible for the connection of UML-RealTime capsules and function blocks of the IEC 61131-3 (standard for PLC programming languages). FBAs contain an interface to capsules as well as to function blocks and a description of the mapping between these interfaces. For this description a special FBA-language is provided. The FBA-language is easy to use both to UML-RealTime and to IEC 61131-3 developers, so they can unambiguously express the interface mapping. An important advantage of the FBA-language is the possibility to use it at an early design state of the UML-RealTime system. We explain our concept of FBAs by an application to a realistic manufacturing system.

1. Introduction and Motivation

Today's industrial manufacturing systems have to face the problem of fast changes in demand of products and in the product spectrum. One solution for getting a more flexible structure of production lines is the concept of autonomous and cooperative production units, which is taken from the idea of holonic manufacturing systems [7].
At the University of Essen an assembly line case study is being developed, which consists of 3 autonomous, cooperative assembly robot cells, a part storage, a product storage, a quality control system, and a transport system. An outline of the case study is shown in Figure 1.
The transport system consists of a belt conveyor on which special pallets are mounted and a Programmable Logic Controller (PLC). The PLC is not shown in Figure 1. The special pallets are prepared to take up parts and products for transportation.
In this paper, products are called with upper letters like A, B, and C, and parts are called with lower letters like d, e, and f. Products consist of parts. A product of type A, for example, consists of parts e and d. In our case study, *product A* is an electrical light switch for surface mounting. It consists of the parts switch box and switch button, which we call d and e, respectively.
The part storage is located at the beginning of the production line. It consists of a storage area for parts and a palletizing robot. When an assembly robot needs parts, the transport system carries empty pallets to the part storage, tells the part storage to put the needed parts on the pallet and carries the pallet to the assembly robot cell.
The assembly robot cell consists of a part storage area, a product storage area, an assembly area, the robot, and an industrial PC (IPC). The IPC (not shown in Figure 1) manages and controls the assembly robot cell. The object oriented IPC-program is designed with UML-RealTime. The assembly robot is able to

(1) take parts from a pallet and put it on the part storage area,
(2) assemble the parts to a product, and
(3) put assembled products to a pallet or the product storage area.

When an assembled product should be carried to the quality control system, the IPC-program asks the transport system for an empty pallet and tells it to carry the product to the quality control system. This transport request is described in more detail in section 2.
The quality control system consists of a vision system with camera and image processor. It is responsible for the decision if an assembled product should be carried to the product storage or to the rejects.
The product storage consists of a palletizing robot and a product storage area.
Each assembly robot cell has the same product spectrum. The number of assembly robot cells working simultaneously depends only on the quantity of demanded products. When this quantity exceeds the capacity of all existing assembly cells, it is also possible to extend the manufacturing system with new assembly robot cells or to

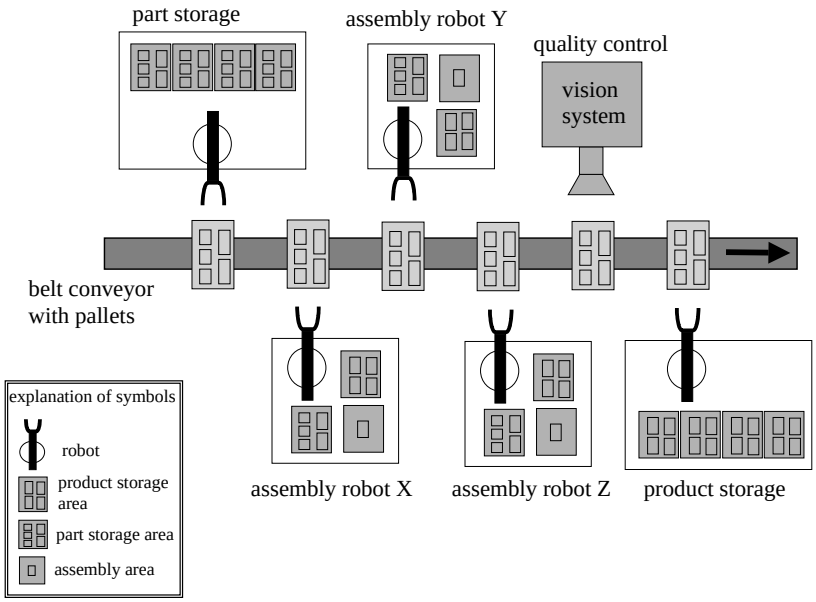

Figure 1. Outline of the assembly line case study

raise the productivity of existing assembly robot cells with improved components.

In cases like mentioned above our approach is to use UML-RealTime for the development of new components. The integration into an existing IEC 61131-3 environment is then modeled with *Function Block Adapters* (FBAs), which we introduce in this paper.

Without FBAs the integration of new components must be done at a very technical level. Depending on the PLC-hardware the communication with PLCs can be established over fieldbus systems, serial interfaces, or digital inputs and outputs. The description of which UML-signals correspond to which PLC software interfaces (IEC 61131-3) is often left out. Instead a developer must extract this information out of low level C code.

With FBAs a developer can describe the mapping of UML-RealTime to PLC software interfaces without knowledge about how the hardware communication is realized.

Every IEC 61131-3 program can be viewed as a Function Block with input and output variables. (In this paper we only discuss Function Blocks defined in IEC 61131-3.) So with a FBA every IEC 61131-3 program can be adapted. FBAs are applied during the object oriented design phase. Technical details are added later in the implementation phase.

To illustrate FBAs by an example we at first pick a small scenario out of our case study (section 2). We assume, that the transport system is described by a Function Block and the assembly robot cell X is described by UML-RealTime. The Function Block interface is discussed in section 3. For better understanding we show only those parts of the interface, which are needed for our example scenario. This is the same for the UML-RealTime interface explained in section 4. In section 5 we introduce our concept of a *Function Block Adapter* (FBA), which is the main contribution of this paper. In section 6 we discuss related work, give a summary and close this paper with an outline of our future work in this area.

2. The Example Scenario

To explain FBAs by an example we use a small scenario out of our case study. For this scenario only a part of the case study shown in Figure 1 is necessary. This part is shown in Figure 2.

We assume that the assembly robot X has produced a product of type A. Now the product must be transported to the quality control. Therefore, assembly robot X sends a transport request to the transport system. This transport request contains the information about the type, sender, and receiver of the product, which is *A*, *assembly robot X*, and *quality control*, respectively. When the transport system receives the request, it needs some time to coordinate the transport request with transport requests of other stations. Much more time is spent on the fact, that a pallet, which can take up the product, has to be carried to the assembly robot.

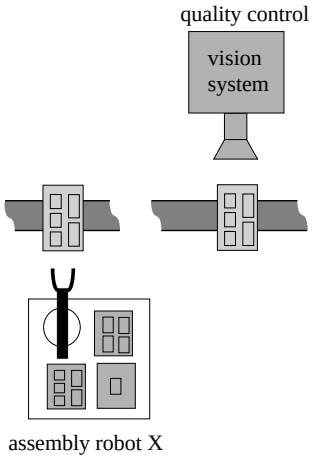

Figure 2. Considered part of the case study

When such a pallet has arrived at the assembly robot X, the transport system tells the assembly robot to put the product on the pallet. This also needs some time for the robot to perform its movements. During this time the transport system may not move the pallet. It has to wait until the assembly robot sends a *pallet free* signal. Then the transport system can carry the pallet to the quality control, where a similar communication takes place.

In the following we concentrate on the communication between the assembly robot and the transport system. This communication consists of the three signals
(1) *transport request*,
(2) *put product*,
(3) *pallet free*.
The next two sections explain how these signals are modeled in Function Blocks (section 3) and UML-RealTime (section 4).

3. The Function Block Interface

As stated in section 1 each IEC 61131-3 program can be viewed as a Function Block. In Figure 3 the Function Block for the transport system is displayed. Its name is *TransportSystem*.
It contains four input variables *Type_IN, StationNr_IN, Take_IN, OK_IN* and four output variables *Type_OUT, StationNr_OUT, Give_OUT, OK_OUT*.
In the PLC system product and part types are called with numbers. Therefore *Type_IN* and *Type_OUT* are integer variables. They are used for product and part numbers in *transport requests*. Stations like robot cells or the quality control system are also called with numbers. In *transport requests* *StationNr_IN* and *StationNr_OUT* contain the information about the participating stations. They are of type integer. Furthermore there are two Boolean variables *OK_IN* and *OK_OUT* which are used for acknowledgements in transport requests. The Boolean variable *Take_IN* is used to signal the transport system with a low-high-edge, that the station given in *StationNr_IN* wishes to put a product or part given in *Type_IN* on a pallet. After a low-high-edge of signal *Take_IN* the transport system awaits the acknowledge with a low-high-edge of *OK_IN*. Then with this acknowledge the destination station for the product or part should be provided in *StationNr_IN*. *Give_OUT* is a Boolean variable used by the transport system to signal the station given in *StationNr_OUT* that it may put the product or part given in *Type_OUT* on a pallet.
The *transport request* signal (1) from our example scenario is shown in Figure 4 by a timing diagram. Such timing diagrams are the normal way for PLC developers to show valid assignments of input and output variables of Function Blocks.
In timing diagrams time is going from left to right. On the very left side the used variables are written. The vertical lines mark special points of time which we need in the following

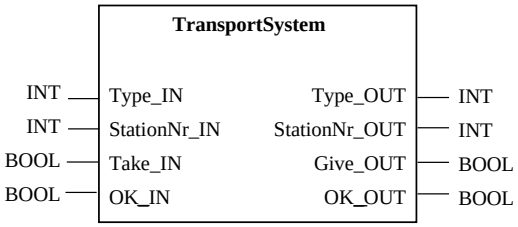

Figure 3. The Function Block interface

for our discussion.
t_1 is the time when the signal transmission of the *transport request* from the assembly robot X to the transport system starts. t_2 is the time when the low-high-edge is applied to *Take_IN*. $t_2 = t_1 + 2$ ms (milliseconds) to make sure that the data on *Type_IN* (number of product A) and *StationNr_IN* (number of assembly robot X) is valid. The transport system acknowledges the data input on t_3 with a low-high-edge on *OK_OUT*. With this edge the input signals are reset by the assembly robot. The time between t_2 and t_3 is determined by the PLC. This is the same for the duration of the *OK_OUT* signal which ends with t_4. On t_4 the assembly robot gives the information about the destination for the product A. t_5 has the same reason like

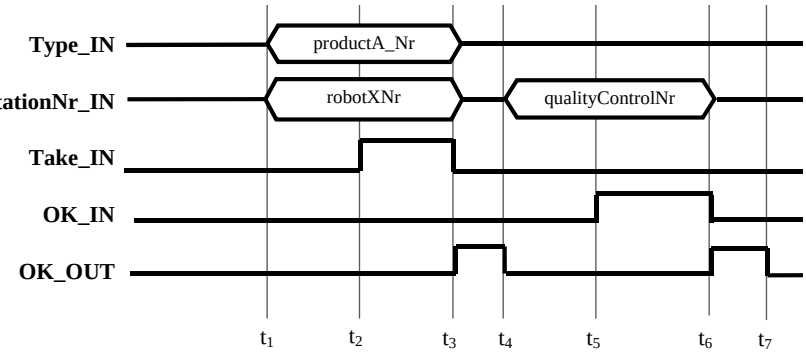

Figure 4. Timing diagram for the transport request signal

t_2. It is to make sure that the input data on *StationNr_IN* (number of the quality control station) is valid. $t_5 = t_4 + 2$ ms. On t_6 the transport system acknowledges the data input with a low-high-edge on *OK_OUT*. With this edge the input signals are reset by the assembly robot. With t_7 the transmission of the transport request is finished.
After this transmission the transport system needs some

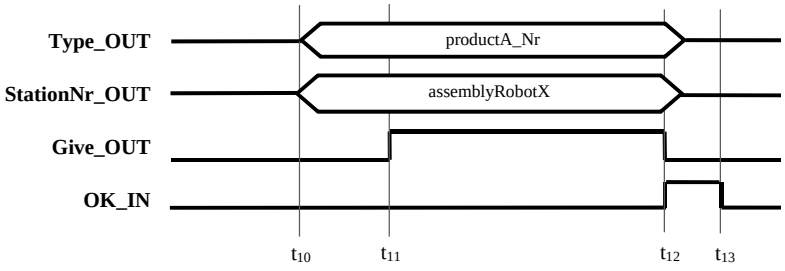

Figure 5. Timing diagram for the put product signal

time to provide a pallet for the assembly robot.
When a pallet is arrived at the assembly robot X, the transport system sends the signal *put product* to the assembly robot like stated in section 2. This is shown in Figure 5. The transmission starts with t_{10} when the output data is set (*Type_OUT* = number of product A; *StationNr_OUT* = number of assembly robot X). With the low-high-edge on *Give_OUT* at t_{11} the signal is sent and valid.
Now the robot needs time to put the product on the pallet. If this is done and the robot arm is out of the collision zone of the transport system, the robot signals *pallet free* at t_{12} for 2 ms. With t_{13} the transmission is finished.
In this section we have explained how a PLC developer would describe software interfaces of the transport system needed for our example scenario. The next section explains the UML-RealTime software interface of the assembly robot X.

4. The UML-RealTime Interface

When object oriented systems are built it is a good idea to introduce a software system architecture at an early design state. An outline of the software architecture is shown in Figure 6.
In this paper we use a special UML dialect called UML-RealTime or UML-RT. An introduction to UML-RT is given in [2]. UML-RT is very similar to ROOM [5] which is in fact the predecessor. A tool for modeling with UML-RT is Rational Rose RealTime. Figure 6 does not use the correct UML-RT syntax because it would go in too much detail to describe the complete architecture. The main idea of the architecture shown in Figure 6 is the use of the mediator design pattern [1], which results in a more loose coupling between system components. The system mediator encapsulates the communication between system components. As a consequence the communication between assembly robot X and the transport system is established over the system mediator. Therefore in this section we explain the communication between the assembly robot X and the system mediator, which defines the assembly robot interface.
We chose UML-RT because of its clear separation of interface and implementation. UML-RT uses special stereotypes to specify interfaces of active objects. Instead of normal classes capsules are applied to model an active class. A capsule communicates over ports with its environment. In Figure 7 two capsules *SystemMediator* and *AssemblyRobot* are displayed. Ports are shown in a new area of the capsule class symbol below the area for operations. Capsule *SystemMediator* has no visible attributes or operations, but two ports called *~transportSystemPort* and *assemblyRobotXPort*. Capsule *AssemblyRobot* has one port called *~transportSystemPort*.

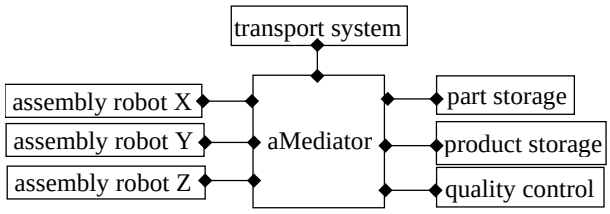

Figure 6. Software system architecture

Over ports only defined signals can be sent or received. The signals of a port are defined by protocols. Protocols are associated to ports. In Figure 7 two associations with stereotype <<port>> are visible which connect port *~transportSystemPort* of capsule *AssemblyRobot* and *assemblyRobotXPort* of capsule *SystemMediator* with the protocol *TransportProtocol*. The *~transportSystemPort* of capsule *SystemMediator* is specified in more detail in

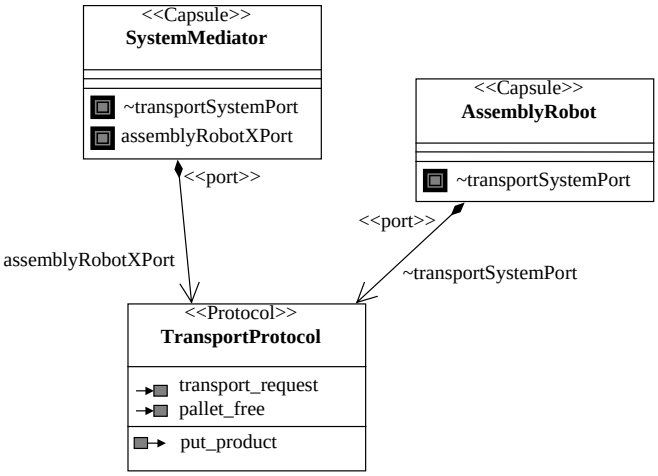

Figure 7. Class diagram

section 5. *TransportProtocol* contains two incoming signals *transport_request* and *pallet_free* and one outgoing signal *put_product*. With this a *SystemMediator* can send signal *put_product* and receive the signals *transport_request* and *pallet_free* over port *assemblyRobotXPort*. If a port name begins with a tilde ~ this port is called conjugated. This means, that the signal direction is inverted. For example an *AssemblyRobot* can send the signals *transport_request* and *pallet_free* and receive signal *put_product* over port

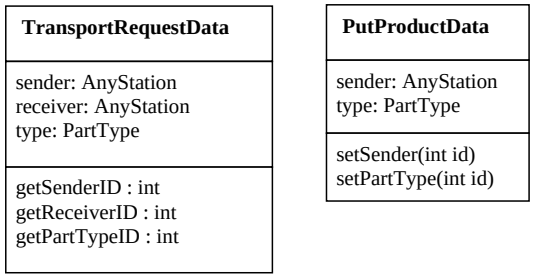

Figure 8. Data classes

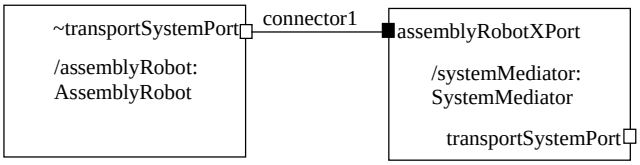

Figure 9. Structure diagram

~transportSystemPort.

A signal is specified with its name and a data class. The data class of signal *transport_request* is *TransportRequestData* (Figure 8). The data class of signal *put_product* is *PutProductData*. *TransportRequestData* contains complex attributes for sender, receiver (destination) and type (product type) of the transport request. To ease the access of identifiers used in the environment of the manufacturing system there are several accessing methods provided (*getSenderID, getReceiverID, getPartTypeID*). The information of class *PutProductData* is set by setting methods (*setSender, setPartType*). Such accessing methods are often used to translate between different nomenclatures. Signal *pallet_free* doesn't need a data class.

Connections between ports are shown in special diagrams called structure diagrams. The structure diagram showing the connection between the *~transportSystemPort* of *AssemblyRobot* and the *assemblyRobotXPort* of *SystemMediator* (connector1) is displayed in Figure 9.

A structure diagram is similar to a collaboration diagram. Both show class roles of objects. In structure diagrams associations are drawn between port symbols. Such associations are called connectors.

A normal port is displayed as a black rectangle and a conjugated port as a white rectangle on the boundary of class roles. With *connector1* a communication between capsules like displayed in Figure 10 is possible.

The sequence diagram of Figure 10 shows the necessary messages to fulfill the requirements of our example scenario. If we forget the relaying time of the mediator, the sending of signal *transport_request* should be in ideal circumstances the time t_2 of the timing diagram in Figure 4. Note that the arrow of this message is single sided. This means, that the message is asynchronous. The sender does not wait for the receiver of the message. The transport system responds at time t_{11} from Figure 5 with the signal *put_pallet*. This is a synchronous message. The sender of the message waits for an acknowledgement. At t_{12} assembly robot X acknowledges with *pallet_free*. The scenario is over.

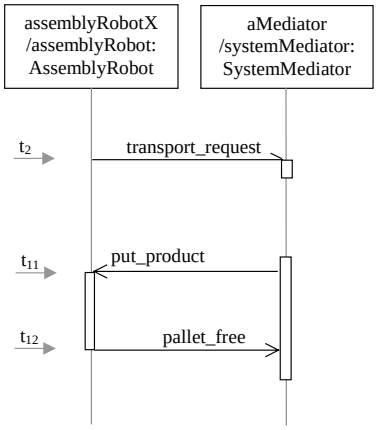

Figure 10. Sequence diagram

The next section explains the missing connection between the capsule *SystemMediator* and the Function Block *TransportSystem* through a *Function Block Adapter*.

5. Introducing a Function Block Adapter

The last two sections explained the interface of the transport system and of the assembly robot. The missing link between them is a *Function Block Adapter* illustrated in Figure 11.

In Figure 11 we show an extended structure diagram with a capsule, a *Function Block Adapter*, and a Function Block. The capsule is the system mediator known from section 4. The system mediator is connected through the port *~transportSystemPort* to the *Function Block Adapter* called *TransportSystemFBA*. *TransportSystemFBA* contains a port called *transportPort* and connection lines to the interface variables of the Function Block *TransportSystem*. In the structure diagram the Function Block *TransportSystem* is instantiated with the name *aTransportSystem*. The *Function Block Adapter* is responsible for the translation of the signals coming from port *transportPort* to assignments of the input and output variables of *TransportSystem*. From the side of the transport system the FBA looks like a Function Block. The variables of the transport system are assigned like explained in Figure 4 and Figure 5. From the view of UML-RT the FBA looks like a capsule. The system mediator can relay the messages to the

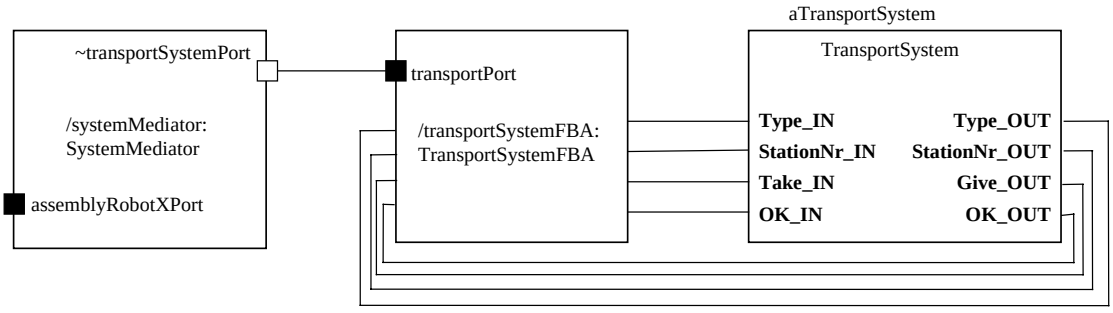

Figure 11. Extended structure diagram with a FBA

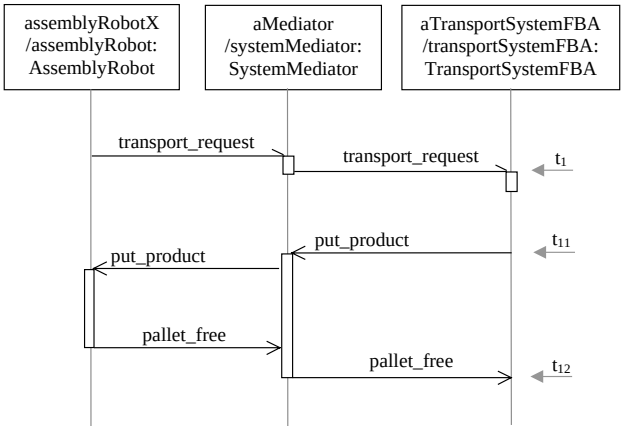

Figure 12. The sequence diagram with a FBA

TransportSystemFBA as it would do it to a transport system capsule which is directly designed in UML-RT. The relayed signals are shown in Figure 12.

Now we can determine the timestamps of the timing diagrams of section 3 more realistic. Of course the FBA cannot forget the delay time between t_1 and t_2 like suggested in Figure 10. So in ideal circumstances the incoming of signal *transport_request* over port *transportPort* is the same time like t_1.

The timestamps t_{11} and t_{12} are also only correct if we forget the relaying time of the FBA. But if we remember, that we are in the object oriented design phase, such thoughts can be moved to the implementation phase. At first we look at how the requirements for the mapping between a capsule and a Function Block can written down.

The class *TransportSystemFBA* is shown in Figure 13. It is of stereotype <<FunctionBlockAdapter>>. This stereotype has the same interface like capsules. Additional attributes are used to represent the input and output variables of the adapted Function Block. These attributes are displayed like normal attributes of UML classes.

The behavior of a FBA is expressed by a special language - the FBA-Language.

5.1 The FBA-Language

The FBA-Language defines operations which are called when signals arrive from a port or from the Function Block. We distinguish between operations for the translation from UML-Signals to Function-Block-Signals (FB-Signals) and operations for the translation from FB-signals to UML-Signals.

In operations of the first category two functions are needed. *Delay(time)* is a function that delays the execution of following commands for the time given as a parameter. *WaitFor(bool, time)* is a function that delays the execution of following commands until the Boolean expression given as first parameter evaluates from false to true. The second parameter is a timeout, which assures that the FBA is not able to hang up. Additionally to these two functions we only need assignments. In assignments access to properties of the FBA class and used data classes is possible. Properties of UML classes are Attributes, Operations, and AssociationEnds. An example operation for the translation of the UML-Signal *transport_request* to the FB-Signal specified in Figure 4 is the following:

```
(1)  ON UML-Signal transport_request PORT
       transportPort
(2)  PRECONDITION OK_OUT = false
(3)  BEGIN
(4)    Take_IN := false;
(5)    OK_IN := false;
(6)    Type_IN :=
         transport_request.getPartTypeID();
(7)    StationNr_IN :=
         transport_request.getSenderID();
(8)    Delay(2ms);
(9)    Take_IN := true;
(10)   WaitFor( OK_OUT, 1s);
(11)   Take_IN := false;
(12)   Type_IN := 0;
(13)   StationNr_IN := 0;
(14)   WaitFor( OK_OUT = false, 1s);
(15)   StationNr_IN :=
             transport_request.getReceiverID();
(16)   Delay(2ms);
(17)   OK_IN := true;
(18)   WaitFor( OK_OUT, 1s);
(19)   OK_IN := false;
(20)   StationNr_IN := 0;
(21)   WaitFor( OK_OUT = false, 1s);
(22) END
(23) POSTCONDITION OK_OUT = false
```

In the following, if we refer to a timestamp t_x these timestamps are given in Figure 4, Figure 5, and Figure 12.

Line (1) starts with "ON UML-Signal" to denote that this

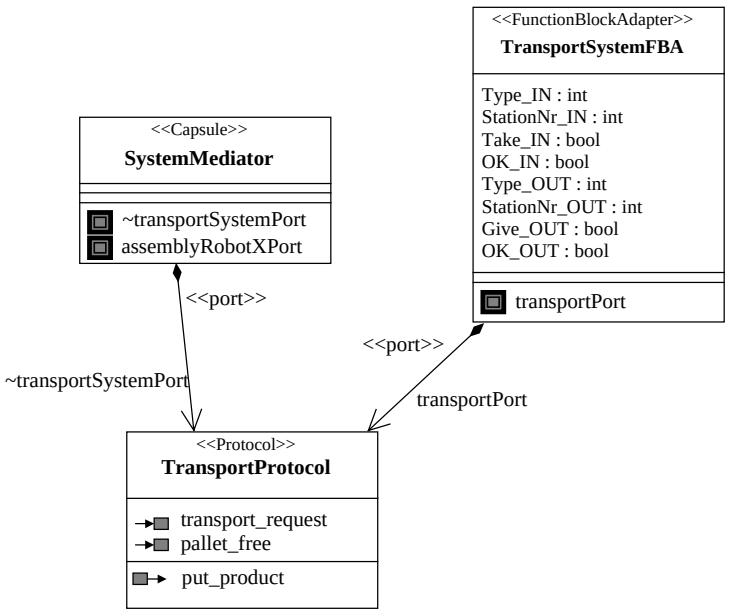

Figure 13. Class diagram with a FBA

operation translates from a UML-Signal to a FB-Signal. "transport_request" is the name of the UML-Signal to translate. Furthermore it is necessary to define the port of the signal. Because we did not restrict a FBA to have only one port, signals with the same name but different meaning can come through different ports. So with "PORT transportPort" the port of signal *transport_request* is defined.

Line (2) is an OCL precondition which is optional.

With line (3) the translation of the message starts. In Figure 4 this is denoted with time t_1.

Lines (4) to (7) assign values to the input variables of Function Block *TransportSystem*. With signal *transport_request* an instance of data class *TransportRequestData* is associated. In our FBA-Language access to properties of this data class instance is given with the use of the UML-Signal name as instance name. So the expression "transport_request.getPartTypeID()" accesses the operation *getPartTypeID* of class *TransportRequestData*. Expressions like this are known from the Object Constraint Language (OCL) of UML. Nevertheless the FBA-Language cannot be a subset of the OCL because OCL is not a programming language. OCL is a pure expression language [6]. Most expressions of our FBA-Language have OCL-Syntax.

Assignments in the FBA-Language take no time. This means, that from line (3) to line (7) there is no time delay.

In line (8) the function "Delay(2ms)" delays time for 2 milliseconds. This is the delay between t_1 and t_2.

At t_2 the input variable *Take_IN* is set with line (9) "Take_IN := true;"

In line (10) the command "WaitFor(OK_OUT, 1s);" waits for the output variable *OK_OUT* to become *true*. If this takes more than a second, an error message is generated and the message translation is cancelled. In normal operation after line (10) the time t_3 is reached.

Line (11) to (13) assign the new values to input variables as given in Figure 4.

With a high-low-edge of *OK_OUT* (line (14)) the destination of the product is given in line (15). After a time delay of 2 milliseconds in line (16) the PLC is forced to take in the destination data with line (17). Line (18) is the same like line (10). Lines (19) and (20) reset the input variables *OK_IN* and *StationNr_IN*. Line (21) waits for the PLC to assure that the message take over is finished. Line (22) denotes time t_7 of Figure 4. The postcondition is in line (23).

The last operation described how the UML-Signal *transport_request* is translated into a FB-Signal. Next we show an operation of the second category for the translation of FB-Signals into UML-Signals. For operations like this additional functions *SendSync(port, send_signal, receive_signal, timeout)* and *SendAsync(port, send_signal)* are needed, which send asynchronous or synchronous messages through ports of the FBA. Furthermore declarations of instances of signals are added which are used in calls of the functions *SendSync* and *SendAsync*. *SendAsync* sends an asynchronous message *send_signal* through port *port*. This asynchronous sending of signal *send_signal* takes no time. If *SendSync* is used instead and *receive_signal* is given as an incoming signal of port *port* and a timeout is set, the function at first sends *send_signal* and then waits for *receive_signal*.

An example of an operation of the second category is the following:

```
(1)  ON FB_Signal Give_OUT
(2)  PRECONDITION (Type_OUT <> 0) &
       (Station_Nr_OUT <> 0)
(3)  SIGNALS
(4)    sig1: TransportProtocol.put_product;
(5)    sig2: TransportProtocol.pallet_free;
(6)  BEGIN
(7)    OK_IN := false;
(8)    sig1.setSender(StationNr_OUT);
(9)    sig1.setPartType(Type_OUT);
(10)   SendSync(transportPort, sig1, sig2, 60s);
(11)   OK_IN := true;
(12)   Delay(1ms);
(13)   OK_IN := false;
(14) END
(15) POSTCONDITION Give_OUT = false
```

The operation starts with "ON FB_Signal Give_OUT" which means that the operation is invoked when *Give_OUT* becomes *true*. *Give_OUT* could also be a Boolean expression. Lines (3) to (5) define to signal instances *sig1* and *sig2*. Line (6) denotes time t_{11}. Lines (7) to (9) set necessary variables. Access to properties of data classes is given by the name of a signal instance, a dot, and the property name. Line (10) is the function call of *SendSync*. The UML-Signal *transport_request* is sent at t_{11}. The function *SendSync* now waits for signal *pallet_free*. After receiving of signal *pallet_free* time t_{12} is reached. Lines (11) to (13) tell the PLC the successful translation of the FB-Signal. At (14) time t_{13} is reached and the operation is finished.

The two operations explained above are typical examples for translation operations of FBAs. All operations consist in their body of the following elements:

- assignments to variables of the associated Function Block
- access to properties of data classes of signals
- calls of the functions
  - Delay(time)
  - WaitFor(bool_expression, timeout)
  - SendAsync(port, send_signal)
  - SendSync(port, send_signal, receive_signal, timeout)

The main purpose of the FBA-Language is to give developers of both UML-RT and IEC 61131-3 a common language for the specification of adapters between components of their models. The FBA-Language is not designed to specify behavior of Function Blocks or of capsules. This means that a FBA does not specify what happens after a signal is translated and sent to a capsule or to a Function Block. This is the reason why we left control structures like *IF THEN ELSE* and loops out of the FBA-Language. If an UML-Signal is such complex that the FBA-Language is not sufficient for the translation to FB-Signals, we prefer to redesign the UML-RT interface instead of extending the language. The reason for this is, that the UML-RT system is applied to an existing system. The UML-RT developer should try to keep his design as conform as possible to the design of the existing system.

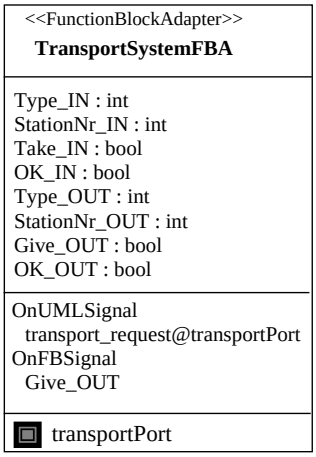

Figure 14. FBA class

Operations of FBAs are documented in the operations area of the class symbol. In Figure 14 the symbol of the *TransportSystemFBA* is given. Below the keyword *OnUMLSignal* the operations of the first category are listed. Their name consist of <signal name>@<port name>. Operations of the second category are listed below the keyword *OnFBSignals*. Their name is the Boolean expression for the FB-Signal.

With FBAs and their language we only specify the "What" should be done to adapt UML-Signals to FB-Signals and reverse. The "How" it is to be done is the responsibility of the developer who implements FBAs.

6. Summary and Future Work

Within this paper we introduced our concept of *Function Block Adapters*. We have shown that with *Function Block Adapters* the integration of systems designed in UML-RT into an existing PLC environment can be easily specified. The specification of a *Function Block Adapter* is completely hardware-independent. It describes only the "What" should be done for the integration and not the "How". This aspect is very important because the "How" is highly hardware-dependent.

An approach related to our FBA-Language is proposed in the Statemate Approach [4]. In Statemate *reactive mini-specs* are used to specify data-driven activities. Data-driven activities are continuously (cyclic) executed, which is expressed with *TICKs* in a *mini-spec*. Conditions are evaluated in *IF THEN ELSE* statements. In our approach FBA-operations are only executed on associated signal events, which is a different semantic than data-driven activities have. For this reason we introduced the notion of a FB-Signal. Conditions on data-values are evaluated with the *WaitFor* function. The decision if conditions are computed continuously or interrupt-driven is left to the implementation. Whereas data-driven activities are suitable for raw sensor data the FBA-Language is easier to use with IEC 61131-3 Function Blocks. We assume that raw sensor data is computed within a Function Block.

With a specification given in the FBA-Language (Section 5.1) a developer has an unambiguous description of the requirements for connecting the UML-RT system to the PLC. Because of the simplicity of the FBA-Language both UML-RT developers and IEC 61131-3 developers can understand and validate the specification.

Currently, we are interested in the development of an implementation framework for *Function Block Adapters*. This framework contains

- an integration process,
- class and Function Block libraries,
- design patterns,
- a FBA-Language parser and compiler,
- a simulation environment for validation purposes.

Furthermore we plan to adapt *Function Block Adapters* to IEC 61499. Function Blocks defined in IEC 61499 distinguish between event input and output signals and data input and output signals. These separation would ease our definition of FB-Signals.

7. References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison Wesley 1995

[2] B. Selic and J. Rumbaugh, *Using UML for Complex Real-Time Systems*, ObjecTime Limited, 1998, http://www.objectime.com/otl/technical/umlrt.html

[3] Programmable controllers - Part 3: Programming languages (IEC 61131-3: 1993)

[4] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts*, McGraw-Hill, New York 1998

[5] B. Selic, G. Gullekson, P.T. Ward, *Real-Time Object-Oriented Modeling*. Wiley, New York, 1994

[6] Object Constraint Language Specification, version 1.1, 1997, http://www.software.ibm.com/ad/ocl

[7] T. Heverhagen, R. Tracht, *Negotiation Scenarios between autonomous Robot Cells in Manufacturing Automation: A Case Study*, Proc. of Tunisian-German Conference *Smart Systems and Devices*, Hammamet, March 2001