Implementing Function Block Adapters

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

Abstract: Function Block Adapters (FBAs) are new modeling elements, responsible for the connection of UML capsules and function blocks of the IEC 61131-3 standard. FBAs contain an interface to capsules as well as to function blocks and a description of the mapping between these interfaces. In this paper we discuss implementation issues of FBAs. While the specification of FBAs is completely platform-independent, we show that different hardware solutions force highly platform-dependent implementation models. In most cases a FBA is implemented in two programming languages - an object oriented language and a language out of IEC 61131-3. While object oriented programs mostly implement an event-driven execution semantic, PLC programs are executed cyclically. Especially this heterogeneous implementation environment was the motivation for developing Function Block Adapters.

1. Introduction and Motivation

Programmable Logic Controllers (PLCs) are widely used for controlling industrial manufacturing systems. The programming of PLCs is normally done in special languages defined in the IEC 61131-3 standard [2]. The increasing complexity of the controlling software for manufacturing systems leads to the need for more powerful specification languages. Latest developments in object oriented technology like UML-RT (successor of ROOM [4]) face this need [1]. But in most cases it is not possible to substitute PLCs in existing plants completely with object oriented systems. Therefore, our approach is to integrate object oriented technology (UML-RT) into an existing PLC-environment in the case of extending a manufacturing system with new components without throwing away the PLC. New components can be for example an Industrial Personal Computer (IPC) which is connected over a fieldbus system to the PLC. We assume, that the IPC program is then designed with UML-RT.

In [5], [7] we introduced a new UML stereotype, the Function Block Adapter (FBA), which is responsible for the connection of UML-RT capsules and function blocks of the IEC 61131-3 standard. 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 understand both to UML-RT 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-RT system.

In this paper we discuss implementation issues of FBAs. We show that different hardware solutions force highly hardware-dependent implementation models. In most cases a FBA is implemented in two programming languages – an object oriented and an IEC 61131-3 programming language. While object oriented programs mostly implement an event-driven execution semantic, PLC programs are executed cyclically. A closer look into cyclic program execution is given in section 2.2. Especially this heterogeneous implementation environment was the motivation for developing Function Block Adapters. First some example requirements are given in section 2. Section 0 gives a short introduction into Function Block Adapters by an example based on section 2. In section 4 two possible hardware solutions are discussed in principle. As an intermediate step towards implementation section 5 introduces an abstract execution model the example FBA. A hardware solution with a fieldbus of type Profibus-DP and a PLC of type S7-300 is discussed in section 6. Section 7 closes this paper with a summary and outlook.

2. Example Requirements

Assuming that there is a PLC on which runs a function block called *MyFB* like shown in Figure 2. It contains three input and three output variables. The Boolean variables *B, C, E*, and *F* are used to provide trigger-events to and from the function block. The data given in *A* is interpreted by *MyFB* depending on the value of *B*. *MyFB* only provides valid data in *D*, when *E* is *true*. Section 2.2 discusses this protocol in more detail.

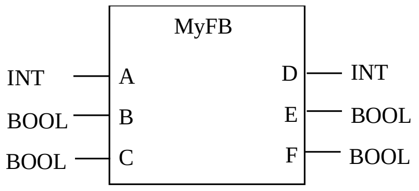


Figure 2. FB interface

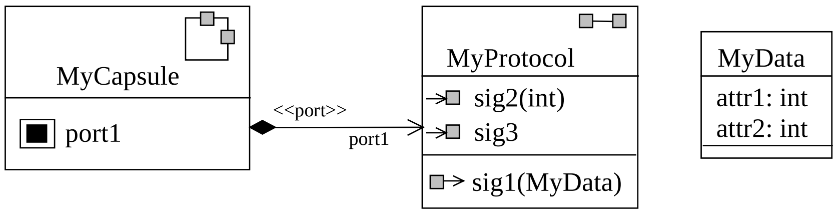


Figure 1. Capsule interface

Furthermore we assume that there is a new application being developed which is designed in UML-RT. This application contains a capsule called *MyCapsule* like shown in Figure 1. *MyCapsule* has to send and receive the signals of protocol *MyProtocol* to and from the function block *MyFB*. This protocol is implemented in *port1*. The mapping from *port1* to the interface of *MyFB* will be explained in section 0.

2.1 The Protocol *MyProtocol*

Figure 3 describes the protocol by a statechart. Initially the protocol is in state *Sig1_or_sig2* if no signal is being transmitted. Transmission of *sig1* is expressed by a transition from state *Sig1_or_sig2* to state *Sig1_or_sig2*. After sending of *sig2* the protocol is in state *Wait_for_sig3*. In this state only sig3 is being able to sent.
State *Sig1_or_sig2* is left by two transitions. If the signals for both transitions arrive at the same time only one transition fires. The selection algorithm is priority-based. In this example the priority of *sig1* is higher than the priority of *sig2*.

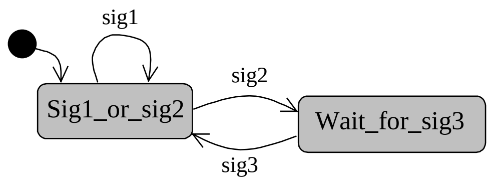
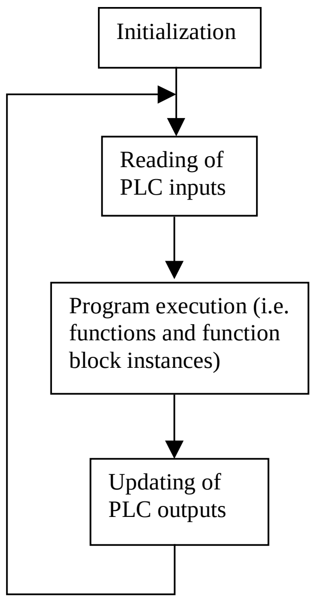


Figure 3. Protocol state machine for *MyProtocol*

2.2 The Protocol of Function Block MyFB

The execution semantic of function blocks is different from the one of capsules. Normally they are executed cyclically. Figure 4 illustrates how program elements are executed within a PLC. After initialization a cycle is started which consists of reading of PLC inputs, program execution, and updating of PLC outputs. The cycle time is mainly determined by the program execution time. Programming languages of IEC 61131-3 contain no statements like waiting for events. If a PLC-programmer implements a waiting or endless loop, the PLC operating system recognizes the loop and refuses the execution. This behavior forces a PLC programmer to a special kind of programming style like explained in section 6.


Initialization
Reading of PLC inputs
Program execution (i.e. functions and function block instances)
Updating of PLC outputs
Figure 4

Instead of having in mind the cyclic execution of *MyFB* it is easier to describe its behavior by a timing diagram given in Figure 5. The timing diagram is divided into three sections by dashed lines. The first section shows how *MyFB* reads sequentially two values from *A*. In this paper we call this FB-Signal *B*, because *B* triggers *MyFB* to read a value from *A*. With a raising edge in *F MyFB* acknowledges that *A* was read. This shall correspond to

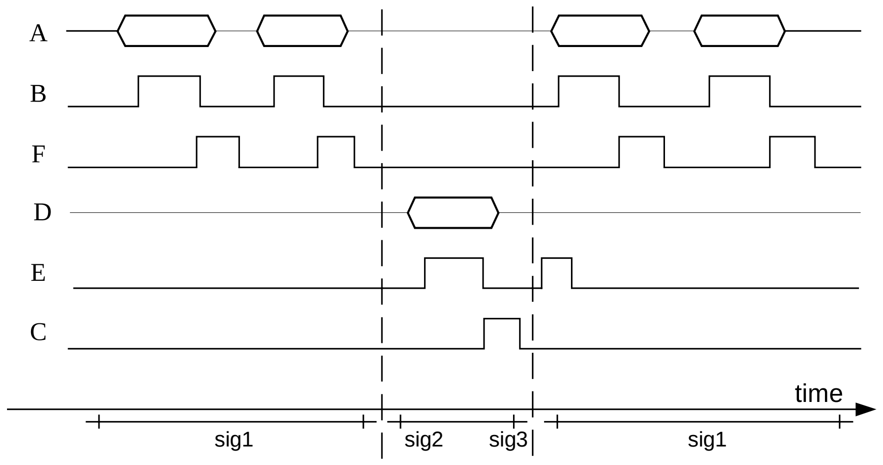


Figure 5

sig1 of *MyProtocol*. The second section describes how MyFB provides some data given in *D* for another function block. We call this FB-Signal *E*, because *E* is the trigger-variable for other function blocks to read a value from *D*. *MyFB* awaits a raising edge in *C* to get an acknowledgement. This shall correspond to the combination of *sig2* and *sig3* of *MyProtocol*. Section three of Figure 5 shows that FB-Signal *B* has higher priority than FB-Signal *E*. FB-Signal *B* can interrupt FB-Signal *E* only until the raising edge of *C* is being reached.

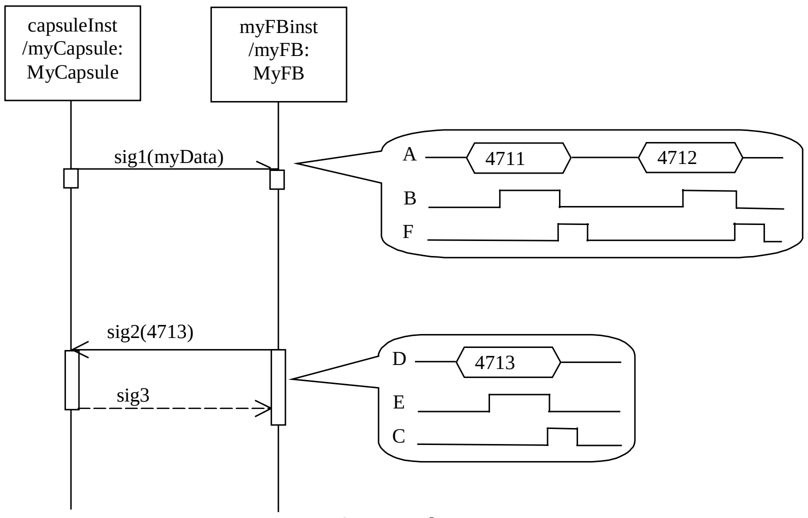


Figure 6. Example interaction

3. Function Block Adapters

Figure 6 shows a possible interaction between an instance of *MyCapsule* called *capsuleInst* and an instance of *MyFB* called *myFBInst*. At first the signal called *sig1* is sent from *capsuleInst* to *myFBInst*. *Sig1* contains an instance of the data class *MyData*: myData.attr1 = 4711; myData.attr2 = 4712.

Of course the function block *MyFB* cannot receive the UML-Signal without a translation into a FB-Signal. The legend which is attached to *sig1* in Figure 6 shows the assignments of the function block variables, which are needed to give the information of UML-Signal *sig1* into the function block *MyFB*. *MyFB* reads the values of *attr1* and *attr2* as a sequence in the input variable *A*. Input variable *B* is used to signal *MyFB* that valid data is assigned to variable *A*. With the output variable *F MyFB* acknowledges the inputs of variables *A* and *B*.

The second signal *sig2* is sent synchronous. This means that the sender (*myFBinst*) waits for an acknowledgement (*sig3*). Graphically an asynchronous message is displayed by a single sided arrow and a synchronous message by a double sided arrow. The answer to a synchronous message is denoted by a dashed arrow.

The data of *sig2* is given in the output variable *D* of *MyFB*. With output variable *E MyFB* tells that the content of *D* is valid. In input variable *C MyFB* awaits the acknowledgement.

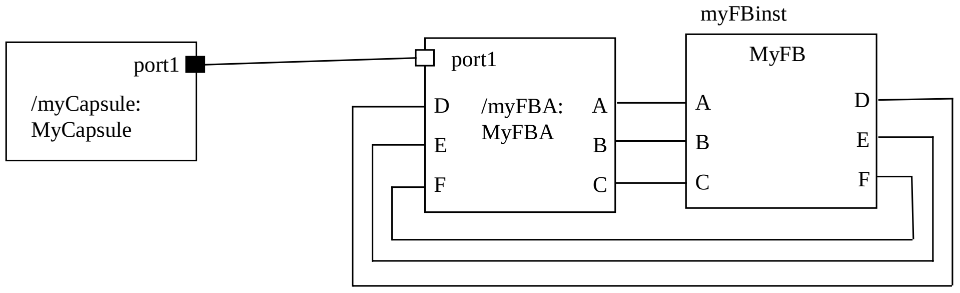
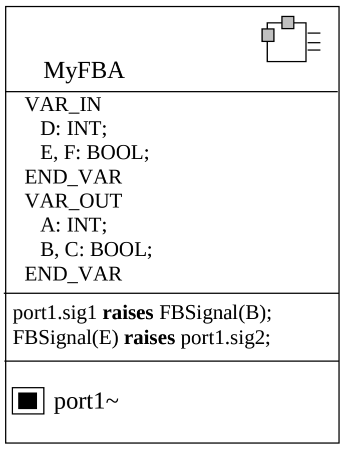


Figure 8. Extended structure diagram for MyFBA

The translation of the timing diagrams into UML-signals is done within Function Block Adapters. Sections 3.1 and 3.2 explain the structure and the behavior of the FBA called *MyFBA*.

3.1 Structure

A FBA is a stereotype of a UML class which contains all properties of a capsule. A FBA uses ports to establish connections to other capsules.
Additionally FBAs define interface variables for the communication with function blocks. A FBA can graphically displayed in an extended structure diagram (Figure 8). The FBA *MyFBA* contains a port *port1~* which is connected to *port1* of *MyCapsule*. Interface variables of *MyFB* which are input variables like *A*, *B*, and *C* are output variables of *MyFBA*. Interface variables of *MyFB* which are output variables like *D*, *E*, and *F* are input variables of *MyFBA*.
The class symbol of *MyFBA* is given in Figure 7. The declaration of the interface variables has the same syntax like in IEC 61131-3 for function blocks. Ports are displayed like ports of normal capsules. Connections to other capsules or function blocks are only shown in the extended structure diagram.



MyFBA

```
VAR_IN
  D: INT;
  E, F: BOOL;
END_VAR
VAR_OUT
  A: INT;
  B, C: BOOL;
END_VAR
```

port1.sig1 **raises** FBSignal(B);
FBSignal(E) **raises** port1.sig2;

■ port1~

Figure 7. Class symbol for MyFBA

The second list compartment of Figure 7 shows two operations of *MyFBA*. These operations are investigated in the next section. Keyword *raises* maps corresponding UML-Signals to FB-Signals and vice versa. This mapping is used by the priority-based selection algorithm for conflicting transitions (section 5).

3.2 Behavior

The behavior of FBAs describe how the translation between the function block interface and the capsule interface is done. For this a special language is provided – 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 *sig1* to the FB-Signal *B* is given in Figure 9.

```
ON_UMLSignal (s1: port1.sig1)
BEGIN
  A := s1.attr1;
  B := true;
  WaitFor( F, T#1s);
  B := false;
  A := 0;
  WaitFor( F = false, T#1s);
  A := s1.attr2;
  B := true;
  WaitFor( F, T#1s);
  B := false;
  A := 0;
  WaitFor( F = false, T#1s);
ON_Exception
  B := false;
END_ON_UMLSignal
```

Figure 9. Translation operation for *sig1*

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(send_signal, receive_signal, timeout)* and *SendAsync(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*. This asynchronous sending of signal *send_signal* takes no time. If *SendSync* is used instead and *receive_signal* is given as an incoming signal and a timeout is set, the function at first

```
ON_FBSignal(E)
SIGNALS
  s2: port1.sig2;
  s3: port1.sig3;
BEGIN
  s2 := D;
  SendSync (s2, s3, T#1s);
  C := true;
  waitFor (E=false, T#1s);
  C := false;
ON_Exception
  C := false;
END_ON_FBSignal
```

Figure 10. Translation operation for FB-Signal *E*

sends *send_signal* and then waits for *receive_signal*. An example of an operation of the second category is given in Figure 10.
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(send_signal)
    - SendSync(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.

4. Hardware Solutions

When implementing a FBA the following points have to be considered:

a) How are the Function Block variables synchronized with the FBA variables?
b) How are the translation operations of FBAs invoked?
   The problem here is the invocation of the translation operations of FB-Signal. UML-Signals are triggers for transition of FBAs. To this transitions the necessary operations for the translation of UML-Signals can be added.
c) How are the functions Delay, WaitFor, SendAsync, and SendSync implemented?

Answers to this questions depend very on the hardware connecting the PLC and the IPC. There is no standard way of connecting a PLC and an IPC. Some general examples of doing this are the following:

4.1 Hardware solution 1

If the PLC interface is very simple, then digital inputs and outputs are sufficient. In most cases the IPC must be extended with a digital I/O card. In this solution the FBA is

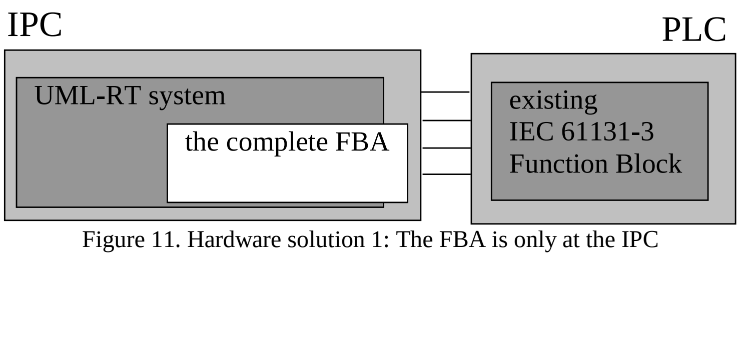


Figure 11. Hardware solution 1: The FBA is only at the IPC

implemented completely at the IPC.
About a) *How are the Function Block variables synchronized with the FBA variables?*
The Function Bock variables can be read and written with the digital I/O card. This can be done by polling or by interrupt techniques.
About b) *How are the translation operations of FBAs invoked?*
Every time a polling function or an interrupt function was invoked, the Boolean expressions of the FB-Signals must be evaluated. If a FB-Signal becomes true, the associated translation operation is invoked.
About c) *How are the functions Delay, WaitFor, SendAsync, and SendSync implemented?*
All functions are implemented and used in the same programming language and environment within the IPC. *Delay* and *WaitFor* could become wait states of a statechart, which are left after a timeout signal of a timer or after the value of a variable has been changed. *SendAsync* and *SendSync* are functions normally provided by the realtime service library of a UML-RT tool.

4.2 Hardware solution 2

A second typical and more important way of connecting a PLC to an IPC is if the PLC uses serial communication over an industrial fieldbus or simply a serial interface like for example RS232 to communicate with an IPC. The IPC uses its existing serial interface or must be extended with a fieldbus interface. The implementation of the FBA then consists of two parts. One part resides at the IPC and the other part at the PLC. Between the two parts a communication protocol must be established within the FBA.

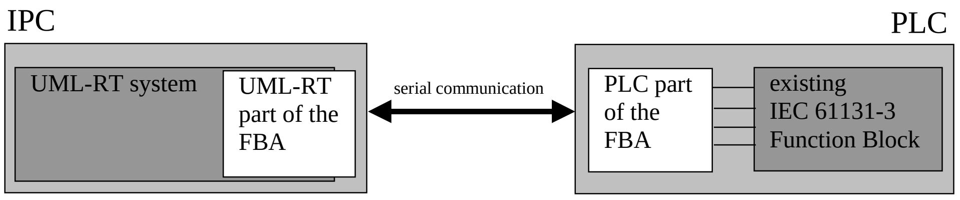


Figure 12. Hardware solution 2: The FBA is distributed over PLC and IPC

About a) *How are the Function Block variables synchronized with the FBA variables?*
For the communication between the PLC and the IPC a special FBA-internal protocol must be developed. Depending on this protocol every changed value of a FB-variable can be sent to the IPC or only meaningful trigger-events. (See also section 5)
About b) *How are the translation operations of FBAs invoked?*
The part of the FBA which is implemented in the PLC is executed in every cycle of the PLC. Each cycle the Boolean expressions of the FB-Signals are evaluated. If a signal becomes true, a message containing all necessary information is sent to the IPC.
Also every cycle the FBA part of the PLC must check if the FBA part of the IPC wishes to send a message.
About c) *How are the functions Delay, WaitFor, SendAsync, and SendSync implemented?*
Delay and *WaitFor* are implemented completely at the PLC part of the FBA in IEC 61131-3 languages. *SendAsync* and *SendSync* are implemented in the IPC part of the FBA. (See also section 6.3)

5. An Execution Model for MyFBA

In this section we explain execution behavior of *MyFBA*. This behavior is a result of a complete FBA-specification given in section 0. It describes when and under which conditions a FBA-operation is processed and in which operational states the FBA may reside. The statechart of Figure 13 shows different abstract states of operation for *MyFBA*. The states are abstract because they must be refined in different ways, depending on the hardware solution (see also section 4). This statechart is an implementation view of the FBA. It doesn't belong to the FBA-Language.

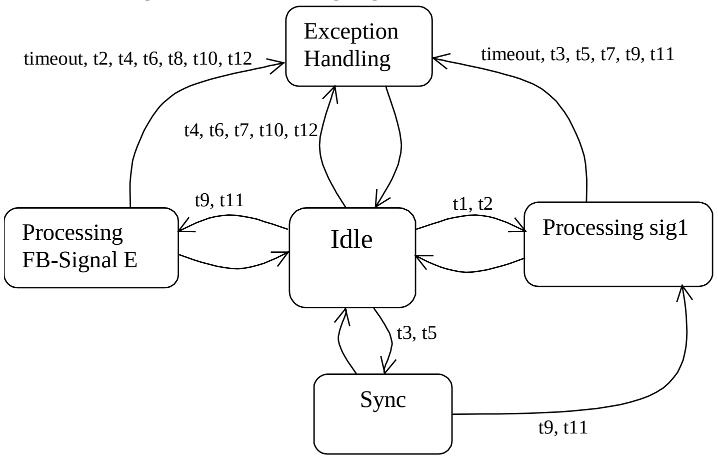


Figure 13. Execution model of MyFBA

The triggers *t1* to *t12* of Figure 13 are defined in Table 1. *MyFBA* has four inputs which may generate events. *Port1* generates the events *sig1* or *sig3* on receiving *sig1* or *sig3*. The input variables *D*, *E* and *F* generate a value-changed event, when the value of a variable has been changed. The events of all inputs are always combined by the logical AND-function. For example, trigger *t1* means that no other event than *sig1* has been occurred. The triggers of transitions in Figure 13 are logical OR-connected.

The central state of Figure 13 is *Idle*. If nothing has to be translated the FBA is in this state. A change-event of variable *D* has no effect in this state. The values of all Boolean variables *E, F, B*, and *C* must be *false*. A change-event of *F* or *sig3* is a protocol exception. This results in a state change to *Exception Handling*. The standard exception handling behavior is to raise an exception message of the operating system. In general, the behavior of this state should be user-defined.

On *t1* or *t2 MyFBA* switches from *Idle* to *Processing sig1*. In this state the FBA-operation of Figure 9 is executed. A further *sig1* remains in the

Table 1. Definition of trigger-events

	port1	**D**	**E**	**F**
t1	sig1	-	-	-
t2	sig1	x	-	-
t3	sig1	x	x	-
t4	sig1	x	-	x
t5	sig1	-	x	-
t6	sig1	-	-	x
t7	sig3	-	-	-
t8	-	x	-	-
t9	-	x	x	-
t10	-	x	-	x
t11	-	-	x	-
t12	-	-	-	x

- means no event.
x means value-changed event.
Events on port1 are named like associated signals.

input queue of *port1* until state *Idle* is reentered. If a deadline is reached or an unexpected event occurred, a transition to state *Exception Handling* fires. In this transition the statements after *ON_Exception* of Figure 9 are executed.
On *t9* or *t11 MyFBA* switches from *Idle* to *Processing FB-Signal E*. In this state the FBA-operation of Figure 10 is executed. A *sig1* remains in the input queue of *port1* until state *Idle* is reentered. With this a kind of extended run-to-completion semantic is reached for the processing states. If a deadline is reached or an unexpected event occurred, a transition to state *Exception Handling* fires. In this transition the statements after *ON_Exception* of Figure 10 are executed.
When in state *Idle* at the same time *sig1* occurs and *E* becomes *true* (*t3* and *t5*), the state *Sync* is entered. Within this transition variable *B* is set to *true*. According to Figure 5 and the priority-mapping given by the FBA (Figure 7) *Sync* is left to *Processing sig1* when *E* is reset.
When implementing hardware solution 2 (section 4.2) every state of Figure 13 must be refined with at least two AND-states, because MyFBA contains two concurrent processes. That's why a FBA-internal synchronization is necessary before one of the trigger-events in Table 1 can be recognized. The next section discusses some technical questions about hardware solution 2 in more detail. Some real time dependent questions, which can be considered at this abstract implementation stage, are discussed in [6].

6. Example for the Fieldbus Profibus-DP and the PLC S7-300

In this section we discuss the hardware solution of section 4.2 realized by a fieldbus of type Profibus-DP and a PLC of type S7-315-DP. The communication between the IPC and the PLC is done over the Profibus-DP. For this the IPC uses a communication processor (CP) called Profibus-CP 5412. The PLC of type CPU315-DP already contains a Profibus-CP.
Like mentioned above, the FBA is implemented in two parts – the capsule part resides at the IPC and the function block part (FB-part) resides at the PLC.

6.1 How are the Function Block variables synchronized with the FBA variables?

Because both the IPC and the PLC are active nodes we need a master-master protocol for communicating over the Profibus. A suitable fieldbus protocol is the FDL (Field Data Link) protocol [8].

```
(1)  void synchronizeAction() {
(2)    ...
(3)    if(SCP_receive(...))
(4)      internalPort.FBSignalE().send();
(5)    ...
(6)  }
```

Figure 14. C++ code fragment

At the IPC the FDL programming interface is provided by a C library with function calls like *SCP_send* and *SCP_receive*. At the PLC the two functions *AG_SEND* and *AG_RECV* are used for FDL-connections. With the FDL-protocol messages can be received either asynchronous or synchronous. The configuration, initialization, and parameter setting of FDL-connections is out of the range of this paper.

```
(1)  FUNCTION_BLOCK MyFBA
(2)  VAR
(3)    my_trig: R_TRIG;
(4)  END_VAR
(5)  ...
(6)  my_trig( E );
(7)  IF my_trig.Q THEN
(8)    ...
(9)    AG_SEND( ... D ... );
(10)   ...
(11) END_IF
(12) ...
(13) END_FUNCTION_BLOCK
```
Figure 15. ST code fragment

As mentioned in section 4.2 the synchronization of the two concurrent processes within MyFBA is done over an internal protocol. This protocol is on top of the FDL-protocol. In our example implementation both sides use polling for recognizing messages of the communication partner. The C++ code fragment in Figure 14 belongs to the capsule-part of the polling mechanism. It calls the function *SCP_receive* to check if the function block part of the FBA wants to send a message with the call of *AG_SEND*. The FB-part of the FBA checks (call of function *AG_RECV*) in every PLC cycle if the capsule part of the FBA wants to send a message with the call of *SCP_send*. The next section explains this aspect again but in more detail.

6.2 How are the translation operations of FBAs invoked?

The translation operations of UML-signals are invoked by the UML-signals itself, after synchronization with the PLC-part was successful.
The FB-signals at first have to be recognized. Then the data of the FB-signal is transferred to the capsule part by a FDL-connection. At the capsule part an internal UML-Signal is generated which triggers the transition which is responsible for the FB-Signal (section 5). This polling is done within the abstract *Idle* state of Figure 13.
The recognition of the FB-Signal is done within the FB-part. The key mechanism is the edge recognition of Boolean expressions. In our example of Figure 10 the Boolean expression consists only of the variable *E*. For edge recognition a function block called *R_TRIG* is provided in [2]. A code fragment of the FB-part of the FBA written in Structured Text [2] is given in Figure 15.
For the explanation of Figure 15 we outline the execution behavior of a PLC in Figure 16. PLC functions are executed in every PLC cycle. At the beginning of a cycle the input

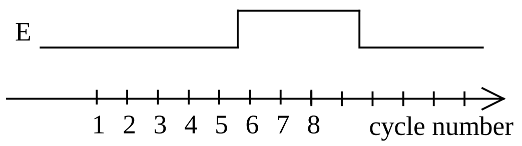


Figure 16. Scanning of input variables of function blocks

variables are read. Line (6) of Figure 15 evaluates in every cycle, if the value of *E* has changed from *false* to *true*. This happens in cycle 6 of Figure 16. Only in this cycle the

output variable *my_trig.Q* is *true*, which is evaluated in line (7) of Figure 15. Then the function *AG_SEND* gives the data of variable *D* to the Profibus system which sends the data over a FDL connection to the Profibus-CP of the IPC.
The time for sending of FDL messages can be greater than the cycle time of a PLC. Furthermore the time interval in which the polling action of the IPC is executed, is in most cases greater than the cycle time of the PLC. This must be considered when FBAs are implemented.
During the design of FBAs we don't need to think about these different cycle times, because a continuous time model is considered. This is a great advantage of FBAs.

6.3 How are the functions Delay, WaitFor, SendAsync, and SendSync implemented?

Delay	For time delays a special function block called *TON* is provided in [2]. (Within the S7-SCL this function block is called *S_ODT*)
WaitFor	The implementation of *WaitFor* is a combination of *R_TRIG* and *TON*. The timer TON is used to generate the timeout.
SendAsync and SendSync	Line (4) of Figure 14 is an example implementation of *SendAsync* with the use of the Rational Rose C++ Realtime Library. For a synchronous message the C++ function *RTOutSignal::invoke()* is provided.

7. Summary and Future Work

Within this paper 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 plattform-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 plattform-dependent.
An approach related to our FBA-Language is proposed in the Statemate Approach [3]. 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 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,
- a modelchecker for verification 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.

8. References

[1] B. Selic and J. Rumbaugh: Using UML for Complex Real-Time System, 1998, http://www.rational.com/products/rosert/whitepapers.jsp

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

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

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

[5] T. Heverhagen, R. Tracht, *Integrating UML-RealTime and IEC 61131-3 with Function Block Adapters*, Proc. IEEE Int. Symp. on Object Orient. Realtime Computing (ISORC2001), May 2-4, 2001, IEEE Computer Society. pages 395-402

[6] T. Heverhagen, R. Tracht, Echtzeitanforderungen bei der Integration von Funktionsbausteinen und UML Capsules, PEARL 2001, Echtzeitkommunikation und Ethernet/Internet (P.Holleczek, B.Vogel-Heuser (Hrsg.)), Informatik aktuell, Springer-Verlag 2001, S. 87-96, in german

[7] T. Heverhagen, R. Tracht, Using Stereotypes of the Unified Modeling Language in Mechatronic Systems, Proc. of the 1. International Conference on Information Technology in Mechatronics, ITM'01, October 1-3, 2001, Istanbul, UNESCO Chair on Mechatronics, Bogazici University, Istanbul, Turkey, pages 333-338

[8] SIEMENS, SIMATIC NET Software NCM S7 for PROFIBUS, Users Guide