A Profile for Integrating Function Blocks into the Unified Modeling Language

Torsten Heverhagen[1], Rudolf Tracht[2], Robert Hirschfeld[3]

University of Duisburg-Essen, Automation and Control, Schuetzenbahn 70,
D-45127 Essen, Germany
[1] torsten.heverhagen@uni-essen.de
[2] rudolf.tracht@uni-essen.de
[3] DoCoMo Euro-Labs, Future Networking Lab, Landsberger Strasse 308-312,
D-80687 Munich, Germany
hirschfeld@docomolab-euro.com

Abstract. In this paper we introduce a new profile for Function Block Adapters (FBAs), which are responsible for the connection of Unified Modeling Language (UML) ports and function blocks of non-UML languages. FBAs provide interfaces to ports, to function blocks, and a description of the mapping between these interfaces. Both UML and function block developers can use a special easy-to-use FBA description language to express these interface mappings in a concise manner. Our FBA language offers the important advantage of high-level descriptions during early phases of the UML development process. This paper proposes a mapping of FBA-semantics to standard UML-semantics. The application of FBAs to function block oriented languages like IEC 61131-3, IEC 61499, or Matlab/Simulink™ is discussed by using the approach of the Model Driven Architecture.

1 Introduction

In engineering disciplines, especially within time driven systems, software languages are often based on function blocks. Examples are languages for programmable controllers [1] or simulation environments like Matlab/Simulink™ [2]. Many commercial software tools in the process industry also use function blocks as program organization units. Despite minor differences the concept of function blocks is the same in all considered function block oriented languages. This concept is described in more detail in section 1.1.

The use of object oriented technologies for event driven systems is widely accepted. Analysis and design of software for event driven systems is done mostly with the UML. Technical systems often consist of both time driven and event driven parts. In such cases it is useful to combine the UML with function block oriented languages. Examples for such combinations are given in this paper and in [3, 4, 5, 6, 7].

It is not our goal to redefine or integrate the behavioral part of function block oriented languages within the UML. This behavioral part is often described by differential equations or by other languages taken from engineering disciplines, which are

complicated to integrate into the UML. One advantage of using function blocks is their separation of external interface and internal implementation. With that it is only necessary to model function block interfaces within UML. The internal behavior can be modeled outside the UML with the help of a function block oriented language.

For the integration of function blocks into the UML it is very convenient to use ports. The port concept is known from [8, 9]. It was first introduced into the UML with version 2.0 [10]. Section 1.2 explains the port concept in more detail. Similar to function blocks, one advantage of using ports is their clear separation of interface descriptions of a class from its internal behavior description. Together with protocol statecharts, ports define protocols, which we call UML protocols throughout the paper. This naming convention is to distinguish UML protocols from protocols of function blocks, which we call FB-protocols in this paper.

The integration of function blocks into the UML is done by providing a special kind of protocol adapters, which we call function block adapter (FBA). They are firstly introduced in [3]. FBAs are responsible for the mapping of UML protocols to FB-protocols. To support a Model Driven Architecture (MDA) like approach this mapping is described independently of platform specific aspects. Section 2 explains the concept of FBAs in more detail from a user's point of view. In [7] the first attempt to define the FBA as a stereotype within version 1.4 of the UML was published. Since the port concept was introduced into the UML version 2.0 it was necessary to rework our definition of the FBA-stereotype resulting in a new profile, which we call *FunctionBlockAdapters*.

In [7] FBAs are only applied to function blocks of IEC 61131-3 (languages for programmable controllers). In this paper we would like to generalize this stereotype to adapt function blocks of other function block oriented languages. We show, that the concept of a function block can also be found in simulation environments like Matlab/Simulink™. Sections 3 and 4 shortly introduce the FBA-profile and discuss the influence of generalizing the profile to other function block oriented languages.

The generalization of FBAs to mathematical languages based on a continuous time model like in Matlab creates a need for comparing FBAs with hybrid modeling techniques. This and a discussion of related work is done in section 5. Finally, section 6 concludes our paper.

1.1 Function Block Oriented Languages

Within this section we introduce our generalized function block (FB) model and give some examples of function blocks in different function block oriented languages. Though the appearance of function blocks differs in function block oriented languages a common structure can be observed. This structure is given in Fig. 1.

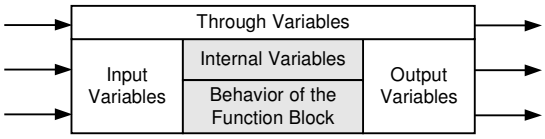


Fig. 1. Structure of a generalized function block

A generalized function block consists of input variables, output variables, through variables, internal variables, and an internal behavior description of the function block. Input variables can only be written from outside of an FB. From inside they can only be read. Output variables can be read and written from inside of an FB and only be read from outside. Through variables are special shared variables. If through variables of different FB instances are connected, they do all access the variable connected to the first input of the chain. Through variables are defined in [1]. They are often called In-Out-variables. If their datatype matches, output variables can be connected to input variables by a connector. This is similar to a connection of ports with matching protocols.

Unlike simple functions, FBs have internal state information that persist the execution of FB instances. The internal behavior can be driven by continuous [2] or discrete time [1], or can be event driven [11]. Common to all FBs is that their interface variables (input, output, and through) continuously provide data values. Communication with other FBs can only be done by assignments of data values to interface variables. This communication model is contrary to the one of object orientation, where objects communicate by message exchange.

Interface variables of FBs cannot be compared to UML attributes. One of the reasons for this is that in object orientation it is not possible to define an attribute, which can be read and written from outside of the owning class and only be read from inside. This is required for the definition of input variables. Another reason is, that attributes of one class cannot be connected to attributes of another class by the means of connectors.

The best counterpart of interface variables in the UML are ports. Both concepts provide export interfaces, import interfaces and encapsulate the internal behavior of FBs and classes respectively. Section 3 discusses the relation between ports and interface variables in more detail. An example of a declaration of interface variables is given in Fig. 2.

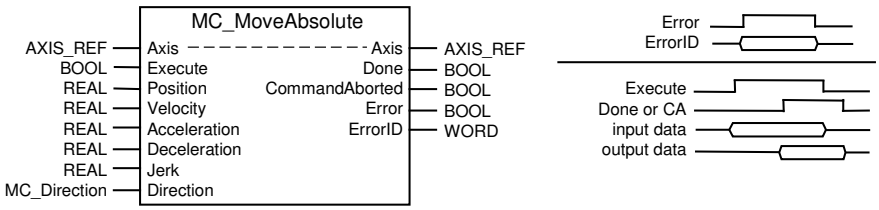


Fig. 2. Function block *MC_MoveAbsolute* and timing diagrams describing its FB-protocol

The FB type *MC_MoveAbsolute* is taken from [12]. It is a standardized FB type responsible for the control of a motion of a motor axis to an absolute position. The notation is conformant to [1]. Interface variables are denoted with input pins (left side) for input variables and output pins (right side) for output variables. Through variables are marked with dashed lines. Variable names are placed inside of the function block and datatypes outside. At the right side of this figure three timing diagrams are shown. The upper one shows that if an error occurs an *ErrorID* is given. Errors can occur even when the axis is not in motion. The middle timing diagram explains that after an

execution of a motion is requested, the motion can be completed (signaled in *Done*) or aborted (signaled in *CommandAborted* (CA)). The so called *input data* is given in the set of input variables *Position, Velocity, Acceleration, Deceleration, Jerk,* and *Direction*. It is only valid, during *Execute* is being set. We define the so called *output data* as the set of output variables *Done, CommandAborted, Error,* and *ErrorID*. Our definition of input data and output data is only to get more convenient notions for the later explanations in this paper. The information given in *Axis* is always valid. With this information several FBs may work on the same axis. For example, an execution of a motion can be aborted by another FB. That's why *CommandAborted* can raise when *Execute* is true.

In Fig. 2 some Boolean variables are used to signal events. This is a common technique for defining FB-protocols. The information given in Fig. 2 can also be used to derive a protocol statechart (see also [12]) or a protocol automaton, which can be used for verification issues [4]. But for FBs it is not necessary to have such kind of triggering variables (Fig. 3).

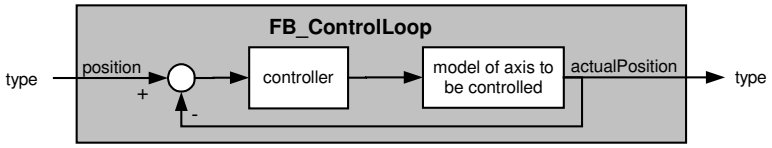


Fig. 3. Function block *FB_ControlLoop*

FB_ControlLoop contains a simplified model of a control loop, which could work inside *MC_MoveAbsolute* of Fig. 2. The input variable *position* is connected to a summation block (denoted as a circle). The second input of the summation block is *actualPosition*. The output of the summation block is the difference between *position* and *actualPosition*. This output is connected to the input of the *controller* block. The internal behavior of the controller is described with differential equations. This is the same for the block called *model of the axis to be controlled*. The output of this block is also the output of the overall block *FB_ControlLoop*. FB-diagrams like in *FB_ControlLoop* are used in control theory and in simulation tools like Simulink [2]. We would like to emphasize that our generalized function block model also works with such languages. Input, output, and through variables as well as internal state information can be observed. The type of inputs and outputs is tool-dependent. For pure mathematics it is of course the set of real numbers.

An actual development in the field of function block oriented languages is IEC 61499 [11]. In this standard the concept of function blocks is extended with event inputs and event outputs. This should ease the application of function blocks to event driven systems. All data inputs and data outputs must be explicitly connected to an event input or event output respectively. An example is shown in Fig. 4.

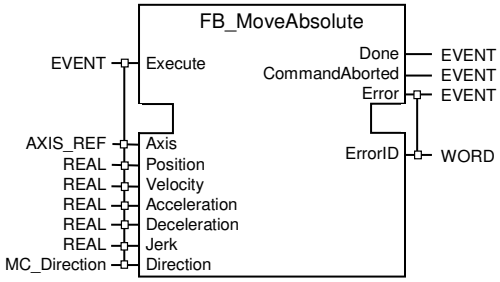


Fig. 4. Example for an FB type conformant to IEC 61499. The FB is separated into an upper and a lower part. At the upper part the event inputs and outputs are drawn. The data inputs and outputs are drawn at the lower part. Because FBs of IEC 61499 don't have through variables, *Axis* is modeled as an input variable and connected to *Execute* like all other data inputs.

The type EVENT can be seen as an abstraction of the Boolean type. Events are stored into Boolean Event Input (EI) and Event Output (EO) variables. Compared to FBs of IEC 61131-3 the interface description of IEC 61499 has richer syntax and semantics. The generalized FB-model of Fig. 1 can be applied to IEC 61499, if event inputs are treated as simple Boolean variables.

In the rest of this paper we use the notation of IEC 61131-3. Differences between function block oriented languages, which are important for our concepts, are discussed where appropriate.

The following section describes a communication interface of a class, which shall be the communication partner of *MC_MoveAbsolute*. For the comparison of both interfaces it is important to emphasize, that FBs communicate synchronously, whereas an asynchronous communication model is assumed between ports.

1.2 UML Ports

Ports had been introduced into object oriented systems in [8]. The first approach for integrating ports into the UML was published in [9]. The port concept we use in this paper is based on [10]. Instead of explaining this concept again, we give an example of its application in Fig. 5.

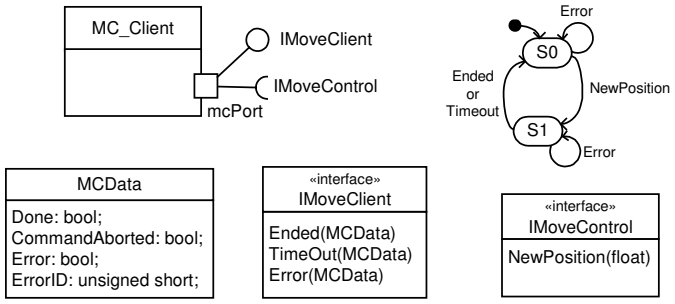


Fig. 5. Classes, interfaces, and a protocol statechart describing the UML protocol

The class *MC_Client* must communicate over *mcPort* with the FB *MC_MoveAbsolute*, which was introduced in Fig. 2. The port provides the interface *IMoveClient*, so it can receive the messages *Ended*, *TimeOut*, and *Error*. All contain a parameter of type *MCData*. *MCData* has attributes to take up the output data of *MC_MoveAbsolute*. We assume, that *MC_Client* is implemented in C++, so the elementary datatypes of C++ are used. The port requires the interface *IMoveControl* to be able to send the message *NewPosition*. The parameter of type *float* contains the new position. We assume, that all other input data is set to constant values, as shown in Fig. 7. The protocol statechart at the top right corner in Fig. 5 defines valid sequences of message transfer over *mcPort*. After *NewPosition* was sent, the messages *Ended*, or *TimeOut* are expected. Errors may occur in every state.

The notion of active objects is introduced in [8] and [9]. Such active objects may interact with their environment using ports. In [10] ports may be added to every object. In this paper we assume, that objects containing ports are active objects. We call a description of the behavior of ports as shown in Fig. 5 a UML protocol. In the next section we explain how communication between ports and FBs can be described using FBAs.

2 Function Block Adapters

A direct connection between a port like that shown in Fig. 5 and an FB of type *MC_MoveAbsolute* is not possible because of their different protocols. For this reason a special protocol adapter, a Function Block Adapter (FBA), was introduced in [3]. FBAs contain ports as well as input and output variables. For the mapping between UML- and FB-protocols a set of translations is defined. Fig. 6 shows an example of an FBA-class.

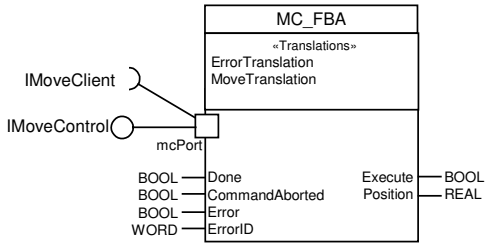


Fig. 6. Function block adapter *MC_FBA*

MC_FBA maps the UML protocol of *clientPort* to the FB-protocol of *MC_MoveAbsolute*. Output variables of *MC_MoveAbsolute* are input variables of *MC_FBA* and vice versa. *clientPort* of *MC_FBA* requires the interface *IMoveClient* and provides *IMoveControl*. The protocol statechart of *clientPort* should be the same like given in Fig. 5. *ErrorTranslation* maps the signal of the input variable *Error* to the *Error*-message of *clientPort*. *MoveTranslation* maps the behavior of the middle timing diagram of Fig. 2 to the messages *NewPosition*, *Ended*, and *TimeOut*. *Abort-Translation* maps the message *Abort* to a signal in the output variable *Execute*.

An instance of type *MC_FBA* is able to connect instances of *MC_Client* and *MC_MoveAbsolute*. If a structure diagram of [10] is combined with the function block language of [1], the connection has a graphical representation like shown in Fig. 7.

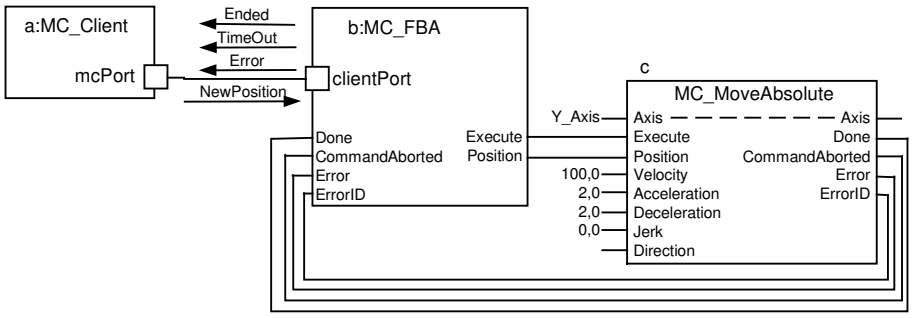


Fig. 7. Structure diagram including an FBA-instance (b) and an FB-instance (c)

An instance *a* of *MC_Client* is connected over *mcPort* with the *clientPort* of instance *b* of *MC_FBA*. The arrows near the connector are only there to illustrate the message directions. Output and input variables of *b* are connected to the FB-instance *c* of type *MC_MoveAbsolute*. As usual in the function block language the instance name in written outside the FB [1]. Most input of *c* are assigned with constant values. The *Direction* input can be left open.

The behavior of *MC_FBA* is defined by its translations. We also call this type of translations FBA-translations. FBA-translations are described by a special language, which we call the FBA-language. This language was designed to be easy to understand by UML-developers as well as by FB-developers. In section 4 we map this language to a subset of statecharts. The language contains syntax for assignments to variables and parameters, wait- and delay-commands, and a send-command. Programs of this language are assigned as tagged values called *translationBody* or *exceptionBody* to FBA-translations. If there is an error during the execution of *translationBody, exceptionBody* is executed. Other tagged values are *trigger* and *signals*. In *trigger*, either a Boolean expression or a message of a port must be provided. The Boolean expression must evaluate to true to trigger the translation. In *signals,* a set of port-messages is given, which are sent, received, or accessed within the translation.

ErrorTranslation	
Trigger:	`Error`
Signals:	`s1: clientPort.Error`

Body:	`s1.Done := Done;`
	`s1.CommandAborted := CommandAborted;`
	`s1.Error := Error;`
	`s1.ErrorID := ErrorID;`
	`send(s1);`

MoveTranslation	
Trigger:	`clientPort.NewPosition`
Signals:	`s1: clientPort.Ended,` `s2: clientPort.TimeOut`

Body:	`Position := trigger;`
	`Execute := true;`
	`waitFor( Done OR CommandAborted, T#1s);`
	`s1.Done := Done;`
	`s1.CommandAborted := CommandAborted;`
	`s1.Error := Error;`
	`s1.ErrorID := ErrorID;`
	`Execute := false;`
	`waitFor( NOT (Done OR CommandAborted), T#10ms);`
	`send(s1);`

Exception:	`s2.Done := Done;`
	`s2.CommandAborted := CommandAborted;`
	`s2.Error := Error;`
	`s2.ErrorID := ErrorID;`
	`Execute := false;`
	`send(s2);`

The deadline stated in the first *waitFor*-statement of the *MoveTranslation* depends on the maximum possible time for the motion of the *Y_Axis* (Fig. 7). The syntax for the time literal *T#1s* is taken from [1]. We decided to use this syntax because it is well-established in the field of automation and control.

We'd like to outline that FBA translations describe the mapping independent of platform-specific requirements. When implementing FBAs it is necessary to use two programming languages, one object oriented and one out of an FB-oriented language. More details on platform dependent issues are given in [6].

Sections 3 and 4 introduce the syntax and semantics of FBAs as a UML profile. In the appendix of this paper we give an example FBA for the function block *FB_ControlLoop* (Fig. 3).

3 A Profile for Function Block Adapters

We call the profile, which defines the stereotype *FunctionBlockAdapter*, *FunctionBlockAdapters*. It contains seven stereotypes which help in the definition of our FBA-concept. Table 1 lists these stereotypes.

Table 1. Stereotypes of the package *FunctionBlockAdapters*. In *Applied to* the metaclass is given, which is stereotyped. *Depends on package* lists the packages which are used by the stereotype.

Stereotype	**Applied to**	**Depends on package**
FBInterface	Interface	Interfaces (from Basic)
FBPort, FBInPort,	Port	Ports (from Common)

Stereotype	Applied to	Depends on package
FBOutPort		
FunctionBlock	Class	Classes (from Common)
FunctionBlock-Adapter	Class	Classes (from Common)
FBATranslation	Class	Classes, CommonBehavior (from Common)

FBInterface is a special Interface, which is needed to model input and output variables of FBs. The interface contains only one operation called *ValueChanged*. This operation has only one unnamed parameter that holds the value of an interface variable of an FB. With OCL this could be written as follows:

```
context FBInterface inv:
 (self.feature->size = 1) &
 (self.feature->forAll( f |
           (f.name = 'ValueChanged') &
           (f.classifier = Operation) &
           (f.parameter->size = 1))
)
```

The datatype attached to the parameter must be one defined by an FB-oriented language. To be able to use such datatypes in UML-interfaces, they must be remodeled within UML as «datatype», «primitivetype» or «enumeration». Different sets of FB-datatypes have no influence on our profile *FunctionBlockAdapters*, so it can applied to different FB-oriented languages.

FB-ports are the direct counterparts of interface variables of FBs. *FBInPort* and *FBOutPort* inherit from *FBPort*. They are signal-ports:

```
context FBPort inv: self.isSignal = true
```

FBOutPorts require one *FBInterface* and *FBInPorts* provide one *FBInterface*:

```
context FBInPort inv:
 (self.required->size = 0) &
 (self.provided->size = 1) &
 (self.provided->forAll( i |
    (i.classifier = FBInterface)))

context FBOutPort inv:
 (self.provided->size = 0) &
 (self.required->size = 1) &
 (self.required->forAll( i |
    (i.classifier = FBInterface)))
```

We assume the behavior of an FB being modeled outside the UML. An FB can be defined as a class only containing *FBPorts*:

```
context FunctionBlock inv:
          self.feature->forAll( f |
            (f.classifier = FBPort) &
```

```
    (f.visibility = #public)
)
```

With this we can model an FB as a UML class as shown in Fig. 8. UML tools with an extension for FBAs could also allow a notation as shown in Fig. 2.

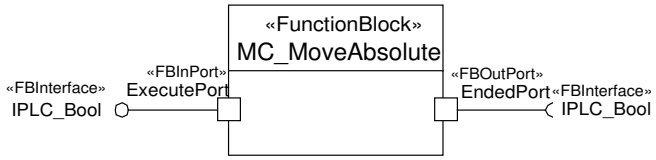


Fig. 8. Notation of FB type *MC_MoveAbsolute* (partially) in a UML-class diagram.

FBAs do have normal ports and *FBPorts*:

```
context FunctionBlockAdapter inv:
          self.feature->exists( classifier = FBPort ) &
          self.feature->exists( classifier = Port &
                                classifier != FBPort )
```

UML-tools with extension for FBAs should support a notation like that shown in Fig. 6 and in Fig. 7.

The behavior of FBAs is described similar to active UML classes by statecharts. Part of our is to hide this statechart from FBA developers. FBA developers should describe the behavior by the means of FBATranslations. UML-tools with extension for FBAs should be able to generate the FBA-statechart out of its FBA-translations. For this reason we provide a mapping from FBA-translations to statecharts. With this, the semantics of FBA-translations is defined as a subset of statechart-semantics. The advantages of doing this is firstly to derive a language which is less complex and easier to understand, and secondly to prevent FBA developers from putting behavior to FBAs, which is not related to the task of translating protocols into each other.

In order to attach FBA-translations to FBAs we defined a tagged value of type *FBATranslation* named *translations*. It is tagged to the stereotype *FunctionBlockAdapter* and holds a set of FBA-translations. Within the class-notation of FBAs this set can be shown in a compartment list as illustrated in Fig. 6.

A set of tagged values is attached to the stereotype *FBATranslation*. They are listed in Table 2.

Table 2. Tagged Values of stereotype FBATranslation

Name	**Type**	**Multiplicity**
fba	FunctionBlockAdapter	1
isOrthogonal	Boolean	1
trigger	Event	1
signals	Signal	1..*
translationBody	String	1
exceptionBody	String	0..1
fbProtocol	ProtocolStatechart	0..1

fba contains the class of the FBA to which the translation belongs. Each translation must belong to exactly one class.

If *isOrthogonal* is set, the translation is executed concurrently to other translations of the FBA. In the example of section 2, all translations are orthogonal.

The event in *trigger* is either a SignalEvent of a port or a ChangeEvent of an interface variable. It triggers the execution of a translation.

In *signals* all port-signals are listed, which must be accessed, received, or sent within the translation. If the trigger was a port-signal, it must be in the list of *signals* under the name trigger.

In *translationBody* and *exceptionBody* a string is given, which conforms to the syntax of the FBA-language. The BNF-grammar of this language is given in [5]. Examples of this language are already introduced in section 2. It is forbidden to use wait and delay statements in *exceptionBody*.

In *fbProtocol* a protocol statechart can be given, which describes all valid sequences of ChangeEvents of interface variables accessed within the translation. In this paper we don't give an example for such a statechart, because it can be derived from the timing diagrams of Fig. 2. In [4] was shown, how such protocol statecharts can be used for verification issues of FBAs.

The next section explains how the syntax of FBA-translations is mapped to statecharts. With this a standard semantics of the UML is defined to FBA-translations.

4 Semantics of FBA-Translations

When thinking about the semantics of FBA-translations some major decisions have to be made. One first decision was if FBA-translations should be mapped to the semantics of operations. In former publications about FBAs we used the notion of FBA-operations instead of FBA-translations. But because wait and delay statements have the semantics of wait states of statecharts, we decided to map the FBA-language completely to statecharts instead of to operations.

Another decision was to use statecharts with or without orthogonal regions. Even though it was promising to base the semantics on roomcharts [8] we decided to use statecharts with orthogonal regions as defined in [9]. This is because even though translations often are orthogonal in their nature, they must access the same set of interface variables. If orthogonal translations are modeled as orthogonal regions of an FBA-statechart, it is easy for them to access the interface variables of the FBA. An example of an FBA-statechart is illustrated in Fig. 9.

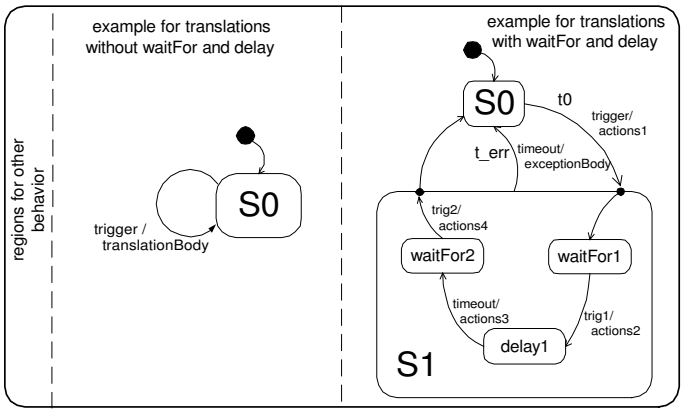
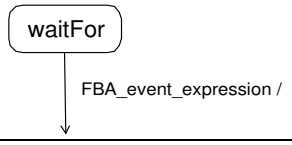
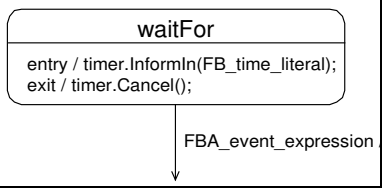


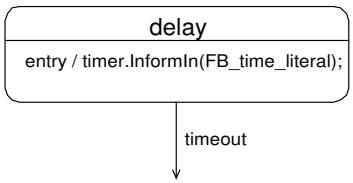
Fig. 9. Example of an FBA-statechart, which could be generated out of FBA-translations.

There is one region for each orthogonal translation. Translations without wait and delay statements can be modeled with only one state. The statements of *translationBody* are actions, which are attached to a transition triggered by trigger. Translations having wait and delay statements are modeled with two states *S0* and *S1*. *S1* has a subchart which is generated according to the *translationBody*. If a timeout is received from an internal timer before a *waitFor*-state is left, the transition *t_err* fires and *exceptionBody* is executed. The FBA-statechart of this figure is not related to the example in section 2.

For the generation of statechart-elements out of the description of FBA-translations, we defined two steps. In the first step all necessary orthogonal regions of an FBA-statechart are created. As a starting point each region must contain a state S0 like in the middle of Fig. 9 or two states *S0* and *S1* like in the right side of Fig. 9. In the second step the behavior given in *translationBody* and *exceptionBody* must be generated. Some rules for this generation are given in Table 3.

Table 3. Semantics of some elements of the FBA-language

Syntax	Semantics
`'waitFor' '(' FBA_event_expression ')'`	
`'waitFor' '(' FBA_event_expression ',' FB_time_literal ')'`	

Syntax	Semantics
`'delay' '(' FB_time_literal ')'`	
assignments, send-statements, operation calls of data classes	combination of «ReadAttributeAction» «WriteAttributeValueAction» «SendSignalAction»

Assignments, send statements, and operation calls of data classes are generated to suitable actions of transitions. The placement of these statements within the sequence of *translationBody* determines, to which transition they are attached. Each region with wait states has its own timer, which is started when a wait state is entered. The timer generates a timeout signal, which is used to leave a *delay* state or to execute the *exceptionBody* in the case of a *waitFor* state is not left in time. If a *waitFor* state is left in time, the timer must be cancelled.

Our semantics definition for FBA-translations uses only a small subset of UML-statecharts. For this subset it is easier to define a formal semantics, which could be used for example for modelchecking [4]. With this we close our discussion of semantics of FBA-translations in this paper. The next section discusses the relation between FBAs, hybrid system modeling and other related work.

5 Comparison to Hybrid Systems and Related Work

As already mentioned FBAs are applied in systems, in which behavior driven by continuous time and behavior driven by events is mixed. Hybrid system modeling approaches face this problem by extending event driven modeling techniques like statecharts with continuous modeling techniques like differential equations [13], [14]. The result of this is a much more complex modeling language than pure statecharts or pure differential equations. To be able to solve differential equations the semantics of statecharts must be extended.

A modeling approach, which adds data flow equations to statecharts of UML, is introduced in [15]. It also introduces data ports, which are similar to our FBPorts. The data flow equations introduced in [15] (like *data trigger connection*) could also be useful to translate FB-protocols into UML protocols. Like FBA-translations this data flow equations can be mapped to the semantics of pure statecharts, since they do not support differential equations.

To distinguish our approach from related work we can outline three major differences. The first one is that FBAs are no hybrid modeling approach. The behavior of FBAs is completely event-driven. There are no equations, which must be evaluated continuously. The second difference is that we do not aim at applying the UML to model behavior driven by continuous time. There are well-established function block

oriented modeling languages like Matlab/Simulink, IEC 61131-3, which have proven to be of great use in this domain. Because FBs that are based on mathematics like in Matlab/Simulink are executable, a MDA-approach for system development is also supported. The third difference is that with FBA-translations we completely substitute statecharts. Instead of enriching statechart syntax, we decided to provide a less complex syntax, which is easier to share between developers of different domains and background. Our experience shows, that in the field of industrial automation and control a huge set of small languages of low complexity is used. These languages are very specialized to devices (robots, vision systems, sensors, controllers) or domains (chemical industry, automotive, telecommunication). Simpler, more dedicated languages usually lead to more concise, less error-prone code, which is an obvious advantage. The disadvantage is the need of integrating components written in different languages. The UML with its language architecture is well-prepared to face the problem of integrating different domain-specific languages, as we have shown in this paper.

6 Summary and Future Work

In this paper we introduce a profile called *FunctionBlockAdapters*. It defines the set of stereotypes, constraints, and tagged values, that are needed to extend UML-tools to support function block adapters. We mapped the semantics of FBA translations to the semantics of statecharts, and defined an FBA-language as a subset of the statechart-language.

The description of behavior of FBAs is carried out using an executable and verifiable language, independently of platform-specific issues. With this a MDA-approach for system development is supported.

The advantage of using FBAs is that they build bridges between purely time driven and purely event driven, port based software components. Each component may use its own language most suitable for its particular application domain. Instead of extending UML-semantics we restricted it to a small subset within FBAs. This is the proposed way of using stereotypes within the UML. The connection of function blocks and UML classes is a frequent task, which makes it worth to be supported by a specialized stereotype.

Currently, we are interested in the development of an implementation framework for function block adapters. This framework contains

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

Another advantage of our approach is simple application to existing UML-tools and FB-environments. So far we have working prototypes integrating PLC-environments and UML tools, as well as Matlab/Simulink and UML tools.

References

1. International Electrotechnical Commission, *IEC 61131 - Programmable Controllers Part 1 – 5*, 1992 Genf

2. Simulink, *Dynamic System Simulation for MATLAB*, The Mathworks, Inc. 2000

3. T. Heverhagen, R. Tracht, *Integrating UML-RealTime and IEC 61131-3 with Function Block Adapters*, Proc. of ISORC 2001, p. 395-402, IEEE Computer Society 2001

4. T. Heverhagen, *Verification of Funktionsbausteinadapters through Modelchecking* (in German), Journal at-Automatisierungstechnik 4/2003, p. 153-163, Oldenbourg 2003

5. T. Heverhagen, *Integration of languages for programmable controllers into the Unified Modeling Language through Function Block Adapters* (in German), PhD Thesis University of Duisburg-Essen, 2003

6. T. Heverhagen, R. Tracht, *Implementing Function Block Adapters*, OMER, GI-Edition - Lecture Notes in Informatics (LNI), P-5, Andy Schürr (Hrsg.), Bonner Köllen Verlag 2001, p. 122-134

7. T. Heverhagen, R. Tracht, *Using Stereotypes of the UML 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, p. 333-338

8. B. Selic, G. Gullekson, P. T. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons 1994

9. B. Selic, J. Rumbaugh, *Using UML for Complex Real-Time Systems*, Rational Software 1999, www.rational.com/products/rosert/whitepapers.jsp

10. OMG document ptc/2003-08-02, *Unified Modeling Language: Superstructure*, OMG 2003, ftp://ftp.omg.org/pub/docs/ptc/03-08-02.pdf

11. International Electrotechnical Commission, *Technical Committee No. 65: Industrial-Process Measurement and Control*, Working Group 6: Function Blocks

12. PLCopen Technical Committee 2 TF, *Function Blocks for Motion Control*, http://www.plcopen.org/forms/motioncontrol.htm

13. R. Grosu, Th. Stauner, *Visual Description of Hybrid Systems*, Workshop On Real Time Programming (WRTP'98), Amsterdam 1998, Elsevier Science Ltd.

14. A. Filippov, A. Borshchev, *Daimler-Chrysler Modeling Contest: Car Seat Model OMER*, GI-Edition - Lecture Notes in Informatics (LNI), P-5, Andy Schürr (Hrsg.), Bonner Köllen Verlag 2001, p. 46-50

15. L. Bichler, A. Radermacher, A. Schürr, *Combining Data Flow Equations with UML/Realtime*, Proc. of ISORC 2001, p. 403-410, IEEE Computer Society 2001

Appendix

For the sake of completeness we provide in this section an example for the use of *FB_ControlLoop* which was introduced in Fig. 3. We assume that a client application is interested in the value of the actual position of the axis. Furthermore it wants to change the value of the input variable *position* (Fig. 3). The client application is called *Simu_Client*. Its protocol is shown in Fig. 10.

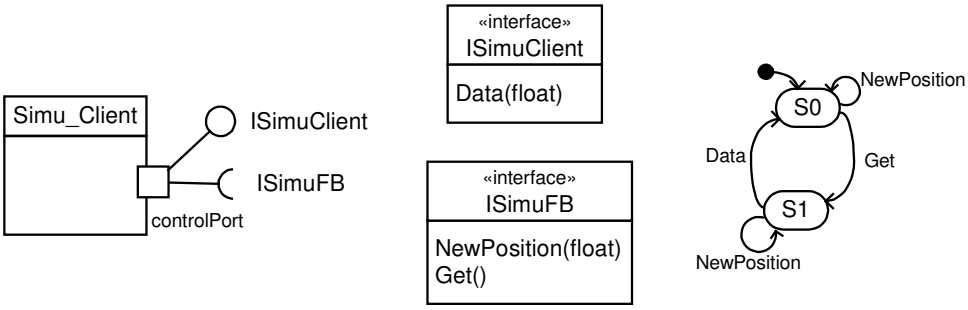


Fig. 10. *Simu_Client* and its protocol.

Simu_Client sends a message *Get* each time, it is interested in the actual position. Within *Simu_Client* this message could be generated with the use of a periodic timer. It is expected that *Get* is answered with the message *Data* which contains the actual position as a parameter. When *Simu_Client* sends the message *NewPosition*, the parameter of *NewPosition* must be assigned to the input of *FB_ControlLoop*.

The FBA *Simu_FBA* was developed to be able to connect *Simu_Client* with *FB_ControlLoop* (Fig. 11 and Fig. 12).

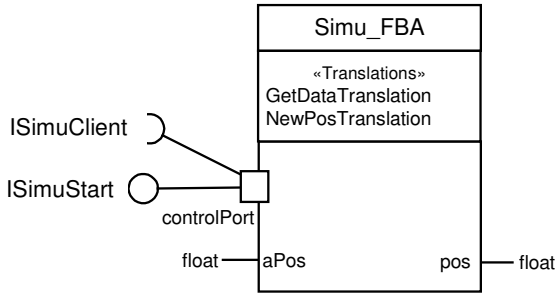


Fig. 11. The FBA class *Simu_FBA*

Simu_FBA contains two translations (*GetDataTranslation* and *NewPositionTranslation*). Both translations are straightforward and need no further explanation.

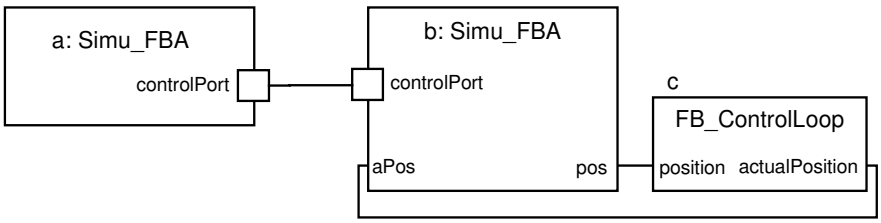


Fig. 12. A possible structure diagram.

```
name = GetDataTranslation
trigger = dataPort.Get
signals = {s1: dataPort.Data}
isOrthogonal = true
translationBody = {
  s1 := pos;
  send(s1);
}

name = NewPositionTranslation
trigger = controlPort.NewPosition
signals = {s1: controlPort.NewPosition}
isOrthogonal = true
translationBody = {
  pos := s1;
}
```