

Using Stereotypes of the Unified Modeling Language in Mechatronic Systems

Torsten Heverhagen, Rudolf Tracht

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

Abstract

The Unified Modeling Language (UML) is the standard design language for developing object oriented applications. It is widely used in the development of complex systems for general-purpose computers. In heterogeneous domains like mechatronics exist a lot of special-purpose programming languages, which are not always easily to map to UML concepts. For such reasons the UML provides an extension mechanism, called stereotyping. This can be used for the mapping of domain-specific languages to the UML. Our approach is to use UML within a mechatronic system for the integration of different specialized design and programming languages. As a placeholder for a system component, which is modeled by such a language, we define domain specific stereotypes. In this paper we compare advantages and drawbacks of using stereotypes by an example stereotype: the Function Block Adapter (FBA). An FBA allows the interoperability between the UML and function blocks of the IEC 61131-3.

1. Introduction and Motivation

The Unified Modeling Language (UML) is the standard design language for developing object oriented applications. It is widely used in the development of complex systems for general-purpose computers. It is supported by most object oriented modeling tools. Typical programming languages for the implementation of UML models are C++ or Java.

In the area of mechatronic systems design environments are specialized to the target hardware or the branch of the target industry. A design environment for a robot cell could for example contain a 3-dimensional layout editor for the working area and some features for planning the movements of the tool center point. On the other hand continuous controllers in process industry are often modeled and simulated with Matlab/Simulink™. A special set of languages evolved for the design and programming of programmable logic controllers (PLCs), which is stan-

dardized in IEC 61131-3. The UML is, in contrast, a completely hardware-independent general-purpose modeling language. When applying the UML to mechatronic systems it turns out that some additional concepts are needed to model such systems. Such concepts can be added by introducing stereotypes. Our approach is to use UML within a mechatronic system for the integration of different specialized design and programming languages. As a placeholder for a system component, which is modeled by such a language, we define domain language specific stereotypes.

Section 2 gives a short introduction into the definition of stereotypes within the UML. Section 3 uses a remote maintenance application to explain example requirements for the integration of a function block into a UML model. In section 4 we provide a solution to these requirements, but we restrict the solution only to contain standard elements of the UML. In section 5 we give a different solution to the same problem by using a special stereotype – the Function Block Adapter (FBA) [8]. A FBA is a placeholder for a function block within a UML model. It can be used to specify the mapping from UML signals to function block signals and vice versa. Section 6 discusses the advantages and drawbacks of both solutions with a special concern on using stereotypes. We close this paper with a summary.

2. Stereotypes

The UML is based on an architecture with four layers:

At the lowest layer are the **user objects** (user data). At this layer instances of classes can be found. For example, the instance `<device_with_serial_number_123>` is of class `Device`.

The next layer is called **model**. This layer contains the classes which define the behavior and structure of the user objects. Class `Device` belongs to this layer. It contains attributes like `serialNr` and operations like `getSerialNr()` and so on. If we speak about a UML model we mean a set of objects at this level.

The **metamodel** layer defines, which modeling elements are available within a UML model. It contains elements

like *Class*, *Operation*, *Attribute*, *Generalization*, *Actor*, *UseCase*, *Component*, ...

The highest layer is the **meta-metamodel**. It defines the language used in the metamodel. Typical objects at this level are *MetaClass*, *MetaAttribute*, *MetaOperation*, ... Instances of *MetaClass* are called metaclasses. Metaclasses belong to the metamodel. For example, object *Class* from the metamodel is a metaclass.

The big advantage of this language architecture is the extensibility of the metamodel. Within the metamodel an existing modeling element can be extended or restricted to specific behavior. This results in a new modeling element, which is called a stereotype of the base element. The mechanism of extending base modeling elements is called stereotyping. The UML provides this mechanism to users of the UML. Another way of extending the metamodel is, for example, to define new metaclasses instead of stereotypes. But this can only be done by the standardization committee of the UML.

With **stereotyping** a base modeling element is extended by new properties and restricted by new constraints. New properties are added by the *tagged value* mechanism. New constraints are added by a formal language which normally is the Object Constraint Language (OCL) [1].

A **tagged value** is a keyword-value pair that may be attached to any kind of modeling element. The keyword is called a *tag*. A tag represents a property like an attribute, an operation or even a completely new kind of property. The value can be a simple datatype value but may also refer to another modeling element.

A **constraint** is a relationship among modeling elements which specify conditions that must be maintained as true. Constraints must be attached to a modeling element. If a constraint evaluates to false, the associated modeling element is invalid. Sections 3 and 4 demonstrate the use of constraints.

2.1 Capsules, Ports, and Protocols

Now we introduce special stereotypes which are of interest to the area of mechatronics. The first stereotype is called Capsule. Fig. 1 shows the example capsule *RemMainBrowser* at the most left position. The capsule belongs to the model layer. The keyword *Capsule* within the quotation marks (called guillemets) «» denotes that *RemMainBrowser* is not a normal class (instance of metaclass *Class*), but a capsule class (instance of stereotype *Capsule*). A user of the UML does not notice whether a capsule is a stereotype or a base element. He or

she simply uses the elements provided by a modeling tool in a toolbar. In such toolbars and in diagrams stereotypes are often presented by special icons instead of keywords like in Fig. 1.

In this paper, if we use the term *Capsule* without a definite or indefinite article we mean the capsule-stereotype at the metamodel layer. A *capsule* or *the capsule RemMainBrowser* or simply *RemMainBrowser* without a definite or indefinite article is an instance of *Capsule* at the model layer. An instance of *RemMainBrowser* at the user object layer is denoted by an article. After section 2 we only care about the model and user object layer. Within this section we concentrate on the relationship between the model and the metamodel layer. The meta-metamodel layer is not affected by stereotyping. The declaration of *Capsule* is partially given in Fig. 2. A more detailed explanation of the concepts of Fig. 2 is given in [1], p. 3-63. A comprehensive description of capsules is given in [2]. The functionality and the meaning of the elements in Fig. 1 is explained in section 3.

Fig. 2 shows a class diagram containing metaclasses and stereotypes. The metaclass *Class* is a subclass of *Classifier* which is in turn a subclass of *GeneralizeableElement*. *ArchitecturalElement* is a stereotype of *GeneralizeableElement*. Consequently, *Capsule* is an *ArchitecturalElement* which stereotypes *Class*. It indicates a class that is used to model the structural components of an architecture specification. To identify if a capsule may be created and destroyed dynamically, a new attribute is tagged to *Capsule* called *isDynamic*. The constraint *{self.isActive=true}* means that capsules represent only active objects maintaining their own thread of control. Capsules run concurrently with other active objects. Additionally there are some more constraints on capsules not shown in Fig. 2. Capsules may only communicate to other capsules through ports with predefined signals. These signals must be defined within a protocol. A protocol is like a capsule also a stereotype of a class. An example of a protocol is given in Fig. 1 called *DataAccessingProtocol*. Between this protocol and *RemMainBrowser* is a directed association called *dataAccessingPort*. Attached with it is the keyword *port*. This means that *dataAccessingPort* is a stereotype of an association called *port*. Through this port a *RemMainBrowser* may send signal *getData* and receive signal *data*. Values of type *Workload* may be attached

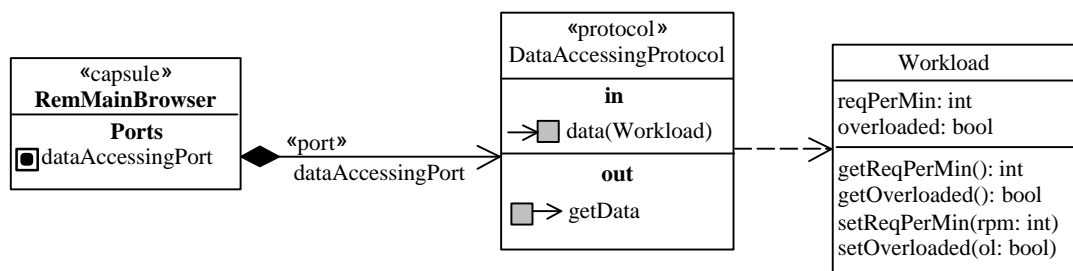


Fig. 1. Example capsule with one port and a protocol which depends on a class (model layer)

with signal *data*. For this reason *DataAccessingProtocol* depends on *Workload*. *Workload* is a normal class with attributes and operations.

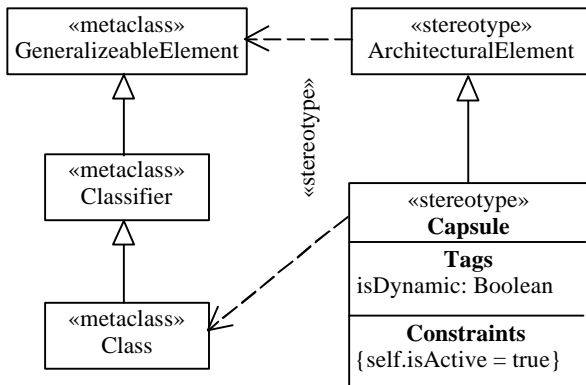


Fig. 2. Declaration of stereotype Capsule (metamodel layer)

All elements of Fig. 1 belong to the model layer. All elements of Fig. 2 belong to the metamodel layer. Relationships between layers are graphically rendered by symbols and optionally by keywords within guillemets. The assignment of *Workload* to *Class* is given by the rectangle around the name. A dashed directed line like between *DataAccessingProtocol* and *Workload* is assigned to *Dependency*. To assign *RemMainBrowser* to *Capsule* instead of to *Class* the keyword *«capsule»* is needed additionally to the rectangle.

The idea of capsules, ports, and protocols came from the field of embedded and real-time systems and were first introduced in [4] as an object oriented design language. In 1999 the first UML tool containing these concepts had been available [5] but until now they have not been within the UML standard. The standard first mentions the capsule-stereotype in [1]. According to [6] these concepts will be part of UML standard version 2.0 which is expected to appear in 2002.

Because to communicate over ports with defined protocols is very natural within mechatronic systems we decided to use capsules rather than normal classes to specify active objects within our models [7]. Furthermore, we developed a stereotype which allows the interoperability between capsules and function blocks.

2.2 Function Block Adapter

In [8] we introduced the FBA for the first time. Our idea is that a FBA looks like a capsule within a UML environment and like a function block within an IEC 61131-3 environment. The first aim is easy to achieve: we declare a new stereotype called *FunctionBlockAdapter* as a subclass of *Capsule*. Fig. 3 shows the declaration of FBA at the metamodel layer. Consequently, FBA inherits all properties of *Capsule*. It is able to communicate with

other capsules through ports. The base modeling element of FBA is *Class*.

IEC 61131-3 does not provide an extension mechanism like stereotyping. This standard is based on a formal grammar instead of on a metamodel architecture. As a result, a function block, which is responsible for communication with a capsule, cannot be distinguished from other function blocks. As a workaround we add the suffix *FBA* to the name of such function blocks. To achieve the same formality as IEC 61131-3 we provided a special language called FBA-language with a formal grammar for FBAs. Within IEC 61131-3 this language is treated as a comment. Within the UML this language is attached by tagged values to FBAs.

An example of a FBA is given in section 5. The complete definition of FBA is out of the range of this paper. In the following sections we concentrate on an example application of a FBA. Further examples are given in [8], [9], and [10]. A discussion about the relationship between design and implementation of FBAs can be found in [9]. A closer look to real-time aspects will be given in [10].

3. The Remote Maintenance Browser

The name *RemMainBrowser* of the capsule in Fig. 1 is an abbreviation for Remote Maintenance Browser. The software component modeled by *RemMainBrowser* belongs to an application for the remote maintenance of a working cell. This application is developed with object oriented languages.

A *RemMainBrowser* is responsible for getting maintenance data out of the working cell and for displaying the data within a browser. Under ideal circumstances the working cell is given as a capsule. This capsule could look like in Fig. 4. It is called *WorkCellController*. It contains the port *~dataPort*. *~dataPort* is an association to *DataAccessingProtocol* (not shown in Fig. 4). The symbol "~" as the first character means that for this port all in-signals of the protocol are out-signals and vice versa. With that, *~dataPort* may be connected to *dataAccessingPort* of *RemMainBrowser*. Such a connection is only established between objects at the user object layer.

Fig. 5 shows a collaboration diagram with an established connection between a *RemMainBrowser* and a *WorkCellController*. *DataAccessingProtocol* specifies the signals, which may be sent and received over this connection, their direction and their sequence.

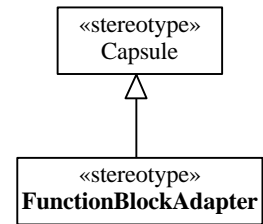


Fig. 3. FBA declaration

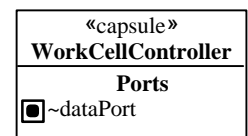


Fig. 4. WorkCellController

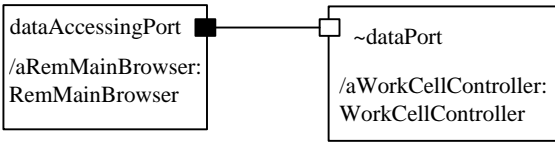


Fig. 5. Collaboration diagram (user object layer)

Allowed signal sequences may be defined within a protocol by a protocol state machine [1], p. 2-175. Fig. 6 shows the state machine of *DataAccessingProtocol*. Initially the protocol is in state *free*. In this state only signal *getData* may be sent. After *getData* the protocol is in state *wait_for_data*. In this state only signal *data* is allowed. The browser updates its data by sending *getData* to the working cell controller and by waiting for *data* from the controller.

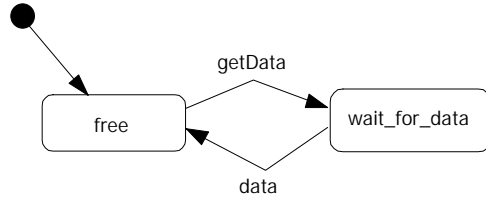


Fig. 6. Protocol state machine

The data value which is sent with signal *data* is of type *Workload* (Fig. 1). Attribute *reqPerMin* contains the actual number of requests per minute, which the working cell has to serve for. Attribute *overloaded* indicates whether the working cell is able to serve for all requests or not. Like usually in object orientation attributes may be accessed and changed by operations like *getReqPerMin*, *setReqPerMin*, *getOverloaded*, and *setOverloaded*. In order to formally specify effects of operations it is possible to write pre- and postconditions with the OCL. Fig. 7 shows example postconstraints for two operations. The sign "=" denotes the logical compare operator and not an assignment. Assignments cannot be expressed by OCL, because it is a language free of side effects. It is not possible to change the state of any object by evaluating OCL expressions. The expression in line (2) means: After executing *setOverloaded* attribute *overloaded* must be equal to *ol*. The body of the operation is left to the implementation. Alternatively we could use natural language like English instead of OCL. The disadvantage would be that the description becomes ambiguous. To avoid this, we use formal languages if possible. The additional use of natural language is necessary to ease communication between developers.

Initially the protocol is in state *free*. In this state only signal *getData* may be sent. After *getData* the protocol is in state *wait_for_data*. In this state only signal *data* is allowed. The browser updates its data by sending *getData* to the working cell controller and by waiting for *data* from the controller.

```
(1) context Workload::setOverloaded(ol: bool)
(2)   post: self.overloaded=ol

(3) context Workload::setReqPerMin(rpm: int)
(4)   post: self.reqPerMin=rpm
```

Fig. 7. Postconstraints for set- and getOverloaded

ble to change the state of any object by evaluating OCL expressions. The expression in line (2) means: After executing *setOverloaded* attribute *overloaded* must be equal to *ol*. The body of the operation is left to the implementation. Alternatively we could use natural language like English instead of OCL. The disadvantage would be that the description becomes ambiguous. To avoid this, we use formal languages if possible. The additional use of natural language is necessary to ease communication between developers.

Until now we collected all information which is needed for communication, if the working cell is modeled by a capsule. If the working cell is modeled by a function block this function block could look like in Fig. 8. *Work-*

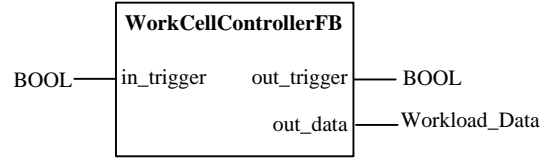


Fig. 8. Function block declaration

load_Data is a user defined datatype given in Fig. 9. It contains two elements. *nr_req_min* shows the number of requests per minute and *overload_flag* signals a possible work overload. The Boolean input variable *in_trigger* triggers the function block to provide actual workload data in *out_data*. In *out_trigger* the function block informs that the data in *out_data* is valid. Fig. 10 shows a timing diagram explaining this behavior in more detail. After *in_trigger* is set, the deadline to set *out_trigger* is 200 ms.

```
TYPE Workload_Data
STRUCT
  nr_req_min: INT;
  overload_flag: BOOL;
END_STRUCT
END_TYPE
```

Fig. 9. Workload_Data

in_trigger is reset when *out_trigger* is set. *out_trigger* and *out_data* remain set for 20 ms. These timing conditions must be known from the function block specification. We use the next two sections to provide different solutions for integrating *WorkCellControllerFB* into a capsule environment. In section 4 we restrict the solution only to contain standard UML concepts. Though the standardization is not finished we treat capsules as standard UML concepts. In section 5 we use an FBA instead of standard UML. We try to provide both solutions as simple and understandable as possible, but our aim is also to achieve the same formality in both approaches.

Fig. 10. Timing diagram

We use the next two sections to provide different solutions for integrating *WorkCellControllerFB* into a capsule environment. In section 4 we restrict the solution only to contain standard UML concepts. Though the standardization is not finished we treat capsules as standard UML concepts. In section 5 we use an FBA instead of standard UML. We try to provide both solutions as simple and understandable as possible, but our aim is also to achieve the same formality in both approaches.

We use the next two sections to provide different solutions for integrating *WorkCellControllerFB* into a capsule environment. In section 4 we restrict the solution only to contain standard UML concepts. Though the standardization is not finished we treat capsules as standard UML concepts. In section 5 we use an FBA instead of standard UML. We try to provide both solutions as simple and understandable as possible, but our aim is also to achieve the same formality in both approaches.

4. Solution 1: Standard UML

In our first solution we add the variables of the function block as attributes to *WorkCellController*. The user defined datatype is modeled by a class. Variable *out_data* is an association to this class (Fig. 11). We assume that these attributes have always the same values like the variables of the function block itself. For the capsule *in_* variables are writeable and *out_* variable are readable. Furthermore the capsule contains an association *actualWorkload* to class *Workload*. In *actualWorkload* the value is stored, which is given in *out_data* and translated to type *Workload*. Operation *setInTrigger* changes the value of

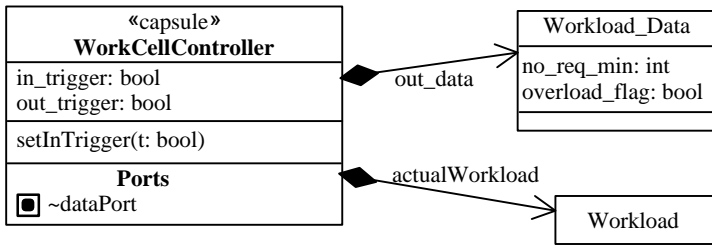


Fig. 11. Class diagram of WorkCellController

in_trigger according to its parameter. It is formally defined in Fig. 12.

```
context WorkCellController::setInTrigger(t: bool)
post: self.in_trigger=t
```

Fig. 12. Postconstraint of setInTrigger

The behavior of *WorkCellController* is specified by a statechart in Fig. 13. Initially the capsule is in state *idle*. When it receives signal *getData* it sets *in_trigger* to *true* and waits for *out_trigger* in state *wait*. If the deadline

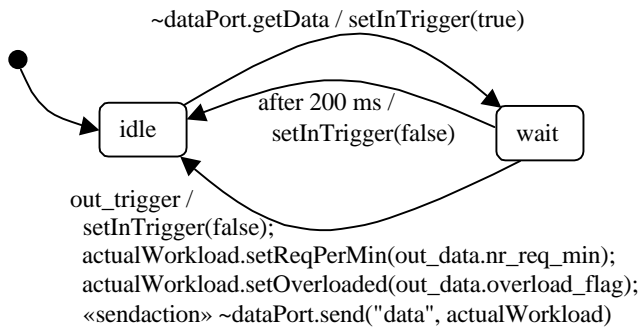


Fig. 13. Statechart of WorkCellController

given in section 3 is reached it sets *in_trigger* to false. The transition *out_trigger* from *wait* to *idle* contains an action sequence. The first three actions are call actions calling operations of the capsule and class *Workload*. The semantics of these actions is given in Fig. 12 and Fig. 7. For the fourth action the semantic meaning is more difficult to express in OCL. We chose to use the keyword *sendaction* which maps the action to metaclass *SendAction* in order to define the semantics of this action. The action language used in statecharts like in Fig. 13 is not defined within UML. By developers the syntax of the implementation language is mostly chosen. But in this example at least two different programming languages are needed: one object oriented language and another out of IEC 61131-3. We decided to choose a semicolon as sequence statement. The keyword *sendaction* is UML style and *~dataPort.send(...)* is similar to a C++ method call in [5].

With this we finished solution 1. The logical mapping of UML signals to assignments of variables is hardware-independent specified. In the next section we do the same with the help of a FBA.

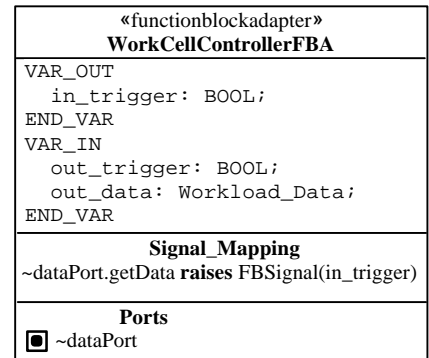


Fig. 14. FBA declaration

5. Solution 2: Using a FBA

A FBA consists of a structural and a behavioral part. The structural part looks similar to capsule *WorkCellController* and is given in Fig. 14. The FBA has the name *WorkCellControllerFBA*. The attribute declaration has the same syntax like the *io_var_declaration* in IEC 61131-3. Section *Signal_Mapping* defines that only signal *getData* may start a communication which is given by a positive edge in *in_trigger* to the function block. What happens after receiving *getData* is specified in the behavioral part (Fig. 15).

```
On_UMLSignal (s1: ~dataPort.getData)
Signals
s2: ~dataPort.data;
Begin
in_trigger := true;
waitFor( out_trigger, T#200ms );
in_trigger := false;
s2.setReqPerMin(out_data.nr_req_min);
s2.setOverloaded(out_data.overload_flag);
sendAsync( s2 );
End
On_Exception
Begin
in_trigger := false;
End
End_On_UMLSignal
```

Fig. 15. Behavioral part of WorkCellControllerFBA

The behavioral part is written in the FBA-Language. This language is based on a formal grammar developed for the FBA-stereotype. For each *Signal_Mapping* an operation must be specified. *WorkCellControllerFBA* only needs one operation *On_UMLSignal (...)*. *s1* can be treated as a signal instance received at *~dataPort*. After *Signals* further signal instances can be declared which are to sent or to received within the operation.

Between *Begin* and *End* the same behavior like in the statechart of Fig. 13 is described. At first *in_trigger* is set to true. Then the execution is stopped until *out_trigger* becomes true or the deadline is reached. After a positive edge in *out_trigger* statements similar to the action se-

quence in Fig. 13 are executed. Properties of classes associated to a signal as an argument are treated as properties of the signal instance within the FBA-Language. The statements *waitFor* and *sendAsync* belong to the language. The correct use of properties like *setOverloaded* must be evaluated with the semantics given in Fig. 7. Statements after *On_Exception* are executed when a deadline is reached or other exceptions occur.

At this point we have concluded the second solution. To be independent of tools it is also possible to express the structural part of Fig. 14 within the FBA-Language.

6. Comparison

Both solutions have the same level of detail. They contain the same information, which is provided to engineers at the implementation phase. Both specify the logical mapping from *DataAccessingProtocol* to the protocol of the function block *WorkCellControllerFB*. Both solutions wrap the function block into a capsule.

The solutions are distinguished by their level of formality within the mapping from attributes of a capsule to variables of a function block. In solution 1 this mapping is stated as a comment in natural language. There is no formal distinguishing mark between a helping attribute like *actualWorkload* and interface variables of a function block. We did not add this information to attributes by tagged values in solution 1, because this would lead to solution 2. In solution 2 a user may declare interface variables using syntax of IEC 61131-3. Technically we tagged strings like "VAR_IN" or "VAR_OUT" to FBA attributes. This information can be used in code generators. Instead of helping attributes it is in solution 2 only allowed to declare signal instances within operations.

Another difference between both solutions is the formality of the behavior specification. Solution 1 uses a combination of a statechart and OCL to achieve an unambiguous specification. Weakness lies within the action language of UML statecharts as discussed at the end of section 4. We did not define a formal grammar for this language, because this would also result in a solution similar to solution 2. In solution 2 we used the FBA-Language to specify behavior. Technically this language is an action language, which is attached to a FBA statechart. The statechart of a FBA is hidden to a user. It can be generated out of the information given in the signal mapping and the FBA-operations. The guideline that a user should not work directly on a FBA statechart is expressed by constraints for FBAs.

Tagged values and constraints can be used to precise semantics of standard UML elements. If there is a problem which often occurs like wrapping a function block users should be forced to specific guidelines. These guidelines may be packed to a new set of stereotypes like capsules, ports, protocols, and FBAs.

7. Summary

UML provides a huge set of general design concepts and graphical languages, which can be applied to almost every domain. Because of its general character it sometimes forces users to specify with natural language or with free-style structured text.

Design and programming languages in mechatronics must be as formal and unambiguous as possible. A lot of specialized languages exist, which fulfill these requirements. The general character of the UML can serve as a basis for integrating these languages. The gap of formality can be closed by the extension mechanisms of the UML.

Within FBAs we achieved a higher formality by restricting capsules to specialized behavior. This is the proposed way for using stereotypes. Furthermore, it is possible to provide specialized graphical presentations for stereotypes. This eases the communication between UML and IEC 61131-3 developers.

8. References

- [1] UML V1.4 draft, OMG document number ad/2001-02-14
- [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] B. Selic, G. Gullekson, P.T. Ward, *Real-Time Object-Oriented Modeling*. Wiley, New York, 1994
- [5] Rational Software Corp., *Rational Rose RealTime Users Guide*, 1999
- [6] Morgan Björkander, *Real-time systems and the UML*, Invited Talk, Workshop Object Oriented Modeling of Embedded Realtime Systems 2 (OMER-2), May 2001, Herrsching a. Ammersee
- [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 (SSD), Hammamet, March 2001, pages 499-504
- [8] 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
- [9] T. Heverhagen, R. Tracht, *Implementing Function Block Adapters*, Workshop Object Oriented Modeling of Embedded Realtime Systems 2 (OMER-2), May 2001, Herrsching a. Ammersee, Report Nr. 2001-03, University of the Federal Armed Forces Munich. pages 11-18
- [10] T. Heverhagen, R. Tracht, *Echtzeitanforderungen bei der Integration von Funktionsbausteinen und UML-RT Capsules*, PEARL 2001, Workshop of the real-time Working Group of the German Informatics Society, 22./23.11.2001, Boppard am Rhein, to appear, in german <http://www.real-time.de/real-time/prog/p2001pvor.html>