

# A Formal Component Concept for the Specification of Industrial Control Systems

Benjamin Braatz<sup>1</sup>, Markus Klein<sup>1</sup>, Gunnar Schröter<sup>1</sup>, and Matthias Bengel<sup>2</sup>

<sup>1</sup> Technische Universität Berlin, Germany  
[bbraatz,klein,schroetg]@cs.tu-berlin.de

<sup>2</sup> Universität Stuttgart, Germany  
Matthias.Bengel@iff.uni-stuttgart.de

**Abstract.** Motivated by the wide acceptance of component based technologies in software development, a component concept for software engineering is applied to modeling in the field of production automation. Taking the modeling of a holonic transport system as an example, it is shown, how function blocks in the sense of production automation can be understood as software engineering components. Thus, the advantages of component based modeling with respect to structuring, exchange and reuse can be transferred to systems in production automation.

## 1 Introduction

Component based technologies for the development of complex software systems, e. g. Java Beans and Corba, found a wide acceptance in today's applied software engineering. These technologies ease the development of large scaled software architectures since the abstraction of the realization of the functionalities to component interfaces minimizes the complexity of architecture building.

In using so called “middleware” it is tried to gain a certain independence of concrete programming languages for the component implementations. But also physical devices with corresponding interfaces can be understood as components, and hence, whole production systems can be described as component architectures.

Not only the improved handling of the structuring of systems, which primarily leads to a reduction of the complexity in the design stage, but also reusability and exchangeability are regarded as important advantages of component based technologies. If a component is implemented once, it is possible to use the same component implementation for another project, if the functionality of the component is, maybe partially, relevant for the new project. This means, the new environment of the component imports functionalities, which are offered by the component. Therefore the services, the component expects from its environment, have to be available in the new context. Moreover, components can be replaced within a single system. In this case, the new component has to fulfill the assumptions of the existing system.

In addition, component structures ease the adaptation of software systems to changed requirements and environment conditions, since the system adaptation

might eventually be limited to single components. If the changed components fulfill their interfaces afterwards, the remaining system components can be left unchanged without any further examination.

In order to specify system components before the actual implementation, a suiting component based specification technique is needed. Existing techniques, especially the UML components in [1], lack a means to express abstract interface specifications. The components in the new UML 2.0 specification in [2] are already a lot more elaborated and have notions of explicit export and import interfaces. Unfortunately, the UML diagram element for an interface still has some restrictions that seem to be inappropriate for a usage in our application field (e. g. it is not intended to instantiate interfaces, which stands in conflict to the ideas presented in Sect. 4). For this reason we will still use a special kind of UML subsystems in this paper, which were developed in [3] to integrate function blocks in the sense of production automation into an object oriented method for the development of production systems presented in [4].

These subsystems are an instance of the abstract component concept presented in [5]. In this approach a component is defined by three specification parts: export, import, and body specification. The export specification describes, which services the component offers to its environment. The body specification describes the realization of the exported service using the imported services of the component. This approach carries the advantages of component based technologies, i. e. abstraction, reusability, and exchangeability, to the design stage of the development process.

For several reasons, it is desirable to have specification techniques available, that are equipped with a formally defined semantics, because this enables the (partial) code generation of implementations for the software part of the system. Beyond this, a formal semantics provides additional advantages. The semantics and correctness of a composed system can e. g. be deferred from the semantics and correctness of its components, provided that the used composition operations are compatible with the semantics. In this paper we will use a specification technique consisting of UML class diagrams and UML statecharts as in [1] and a simple action language based on the notations of OCL. For a number of UML techniques a formal semantics based on transformation systems was developed in [6, 7], which have been summarized and integrated in [8].

In [9] we have already presented first ideas concerning the correspondence of function blocks according to [10] and software specification components. In contrast to [9] we will give a much more detailed discussion in this paper and we show how the used component framework can be extended by a model based semantics.

The paper is organized as follows: In Sec. 2 we show how function blocks in production automation can be modelled using diagram techniques from the UML. In Sec. 3 we interpret these UML function blocks as components in a formal component concept. In Sec. 4 we sketch a model-theoretic semantics for such components. Finally, in Sec. 5 we conclude by giving some directions of future work.

## 2 Modelling Function Blocks with UML Subsystems

In this section we show how function blocks according to the IEC/PAS 61499-1 specified in [10] can be modelled by UML subsystems in the sense of [1]. This approach, using the ODEMA<sup>3</sup> method presented in [4, 11], was originally introduced in [3].

Figure 1 shows the characteristic elements of a simple function block according to IEC/PAS 61499. These elements are mapped to a UML subsystem function block in the following way: In the specification part of a UML function block we model the event and data flow of the services the subsystem offers to its environment (export interface), while distinguished (proxy) elements in the realization part model the event and data flow of services it assumes from its environment (import interface). The rest of the realization part models internal data, execution control and algorithms of the function block by a class diagram, statecharts and action language expressions.

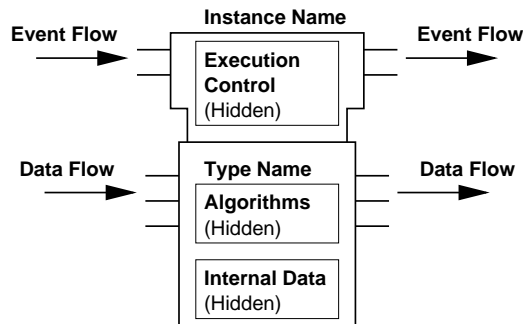


Fig. 1. Function block characterization

The specifications of function blocks shown below are slightly modified parts of a solution for the reference case study “holonic material flow”, presented in [12], within the DFG priority program “Software Specification” (see [13]). This solution was developed in the project IOSIP<sup>4</sup> and can be found under [14].

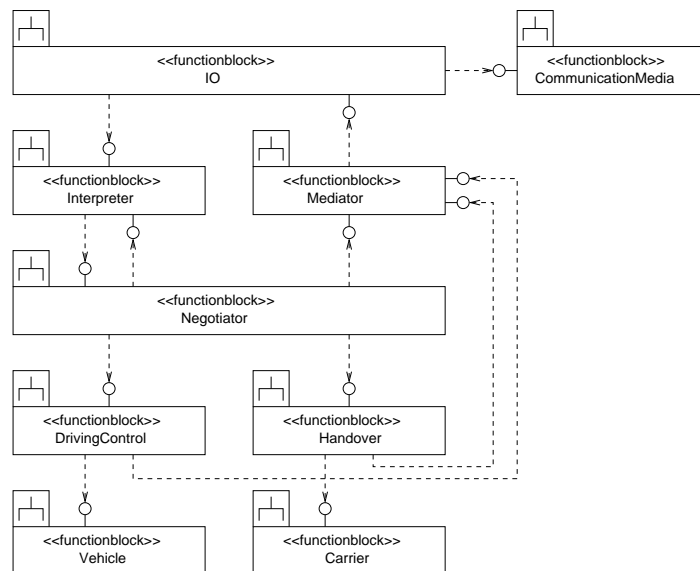
Figure 2 shows the function block structure of the control software of an H-AGV<sup>5</sup>, where function blocks are symbolized as UML subsystems. The function block IO reads out the incoming data stream from the CommunicationMedia. Data packages that have to be send to other H-AGVs are transferred through the IO function block, too. Hence, the function block CommunicationMedia, which is

<sup>3</sup> Object-oriented method for developing technical multi-agent-systems

<sup>4</sup> Integration of object-oriented software specification techniques and their application-specific extension for industrial production systems on the example of automobile industry

<sup>5</sup> Holonic automated guided vehicle

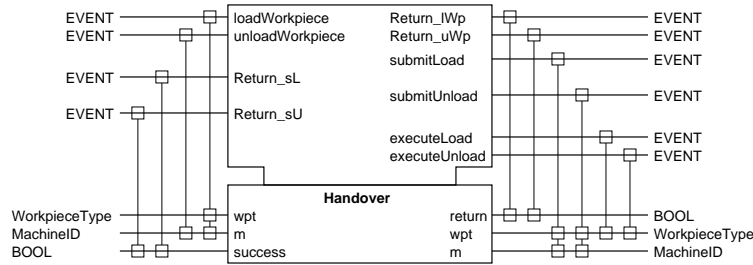
a function block realized by an apparat, has to offer features for receiving and sending data in its interface. The function block **Interpreter** analyzes incoming messages that are relevant for the particular H-AGV. If, for example, a machine status was received, a possibly necessary negotiation would be initiated. For this purpose, a calculation of costs and the composition of outgoing messages would be done in the **Negotiator** function block. These messages would be put into the used communication protocol in the **Mediator** function block, before they reach the I/O function block. If the negotioation ends up with transport order, the controlled H-AGV has to drive to the particular workpiece. This happens under the usage of the **DrivingControl** function block, where the controlling of the driving relevant sensors and actuators in the function block **Vehicle** is implemented. The function block **Handover** realizes the loading and unloading of workpieces between H-AGVs and machines by a carrier apparat in the **Carrier** function block.



**Fig. 2.** Architecture of the H-AGV control software

The function block **Handover**, shown in Fig. 3 uses the sensors and actuators of the workpiece carrier of the H-AGV. Thus, the function block has output events that are used as communication signals for the function block **Carrier**. The small boxes on the lines modeling the event and data flow mark synchronization points. For example, the event `loadWorkpiece` is connected to two data elements, `wpt` and `m` of type `WorkpieceType` and `MachineID`, respectively. The `Return` events are meant as signals that report the termination of a cor-

responding method, where some of them are connected to data flow elements, which means that this method has a return value of that type.

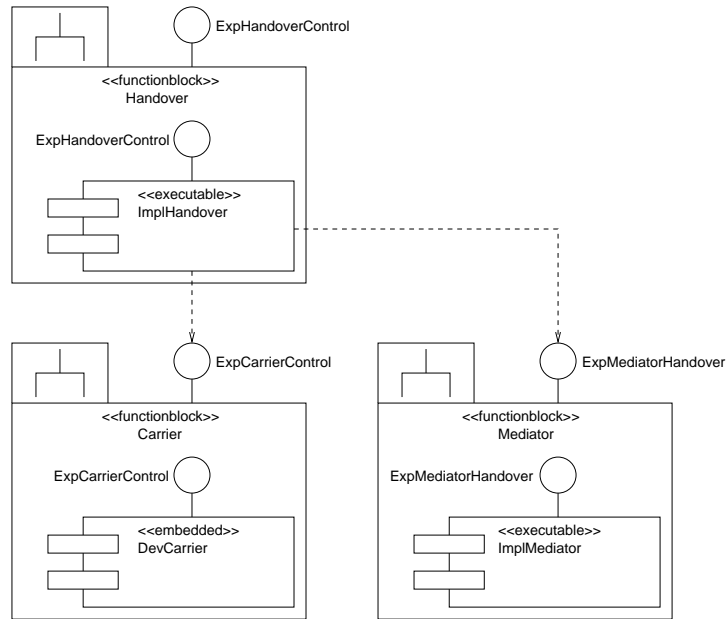


**Fig. 3.** Function block Handover

This representation style might be very useful for realizing function blocks by physical devices, but if a function block is planned to be realized by a piece of software, it should be expressed in a modeling language which is more common in the software engineering world. We will use UML diagrams and the UML based method ODEMA for this purpose together with a mapping of function blocks to UML subsystems originally presented in [3]. Figure 4 shows the structure of the function block **Handover** and the function blocks it relies on to provide its functionality, where the function block **Carrier** is realized by an apparatus and the function block **Mediator** by a software component. This is represented in the diagram by corresponding UML components with the stereotypes `<<embedded>>` and `<<executable>>`, respectively.

In Fig. 5 it is shown, how the event and data flow structure of the function block **HandoverControl** can be represented by UML classes. Since this function block contains a single control component only, we can draw the class diagram directly into the subsystem frame. In general, function blocks can contain more than one control component. The export interface subsumes the methods which are declared in the interface of the corresponding function block with a `<<com>>` stereotype for the communication with embedded subsystems. We use synchronous method calls for the communication between executable software subsystems. Moreover, the dependencies to other subsystems are redirected through interfaces marked with the stereotype `<<ODproxy>>` in order to make explicit the imports of the subsystem, which leads to a self-contained description of the function block.

In Fig. 6 we see the specification of the reactive behavior of the function block by a statechart. We can regard this as an addition to the export interface as well as the body of the function block specification. The statechart allows other functionblocks to call the method `loadWorkpiece`, whenever the carrier is empty, and the method `unloadWorkpiece`, if some workpiece is loaded onto the



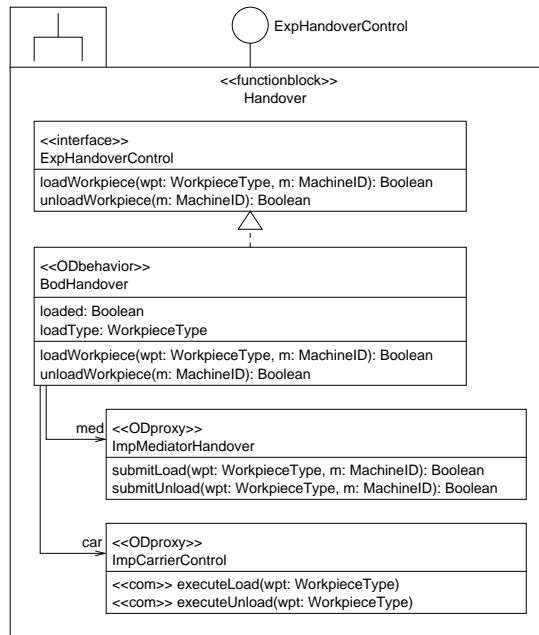
**Fig. 4.** Subsystem structure of the function block **Handover**

H-AGV. Both methods non-deterministically choose to either switch the state or stay in the same state, since this choice depends on the success of the imported submit methods of **ExpMediatorHandover**, which are not visible in the statechart. Other function blocks importing **Handover** have to ensure, that this protocol is respected.

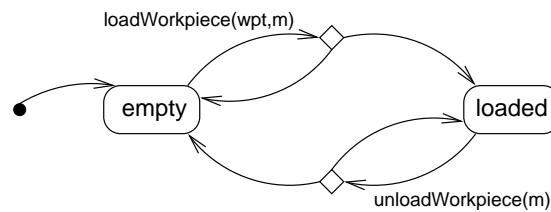
The specification of the realization of the methods is encapsulated in the body specification of the function block. Figures 7 and 8 show such action language specifications for the methods **loadWorkpiece** and **unloadWorkpiece**, respectively, where we use a rather simple ad hoc action language, which is neither part of the UML nor ODEMA at the moment. The description shows the concrete sequence of actions and how other methods, e. g. **submitLoad**, and signals, e. g. **executeLoad**, are used in the method.

An addition to the import specification is the protocol statechart in Fig. 9, which constrains the signals **executeLoad** and **executeUnload** to be sendet alternating. On the other hand the methods of **ImpMediatorHandover** are not further constrained.

In the next section we will show, how function blocks modelled by UML subsystems as presented in this section can be interpreted as components in a formal specification component concept, and which benefits can be drawn from two kinds of semantics for these components. Especially we will see how architectures of function blocks can be flattened by a composition operation based on a transformation semantics for components.



**Fig. 5.** Detailed structure of the single control component in the function block Handover



**Fig. 6.** Statechart of the function block Handover

```

context BodHandover::loadWorkpiece(wpt: WorkpieceType, m: MachineID):
if submitLoad(wpt, m)
then executeLoad(wpt);
      set loaded := true;
      set loadType := wpt;
      return true
else return false
endif
  
```

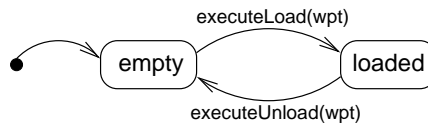
**Fig. 7.** Action language description of the method loadWorkpiece

```

context BodHandover::unloadWorkpiece(m: MachineID):
if submitUnload(loadType, m)
then executeUnload(loadType);
      set loaded := false;
      unset loadType;
      return true
else return false
endif

```

**Fig. 8.** Action language description of the method `unloadWorkpiece`



**Fig. 9.** Statechart of the import `ImpCarrierControl`

### 3 UML Subsystems in an Abstract Component Concept

In this section we shortly introduce an abstract component concept based on the ones published in [15] and [5]. Then we show, how UML subsystems can be interpreted as components in the sense of this concept and which semantical implications arise from such an interpretation.

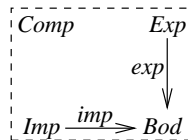
In contrast to software components such as the ones in CORBA, Java Enterprise Edition or .NET, the component concept considered here is concerned with specification components, which are means to structure large specifications into manageable parts. The concept of specification components is orthogonal to the software components in the implementation and deployment structure, i. e. there is not necessarily a bijective correspondence between specification and software components, but there could also be specification components, which specify several software components, and, vice versa, software components, which are specified in several specification components.

A component *Comp* in this framework consists of import and export interface specifications *Imp* and *Exp* and a body specification *Bod*, which is supposed to realize the provisions granted in *Exp* using the requirements stated in *Imp*. Note, that *Imp*, *Exp* and *Bod* are in general supposed to be constructed from the same specification techniques. Hence, the interfaces are not restricted to certain kinds of specifications, unless we explicitly impose such a restriction for an instantiation of the component concept.

In [15] it is additionally assumed, that there is a model-theoretic semantics for the specifications, which induces a model class  $Mod(Spec)$  for each specification *Spec*. An instantiation of this requirement by models suitable for the specification of object-oriented systems is sketched in Sect. 4. A viewpoint concept used to manage the specification of heterogeneous aspects of such systems is introduced

in [16] in this volume, where this concept lies orthogonally to the concepts of parameterization and abstraction realized in components.

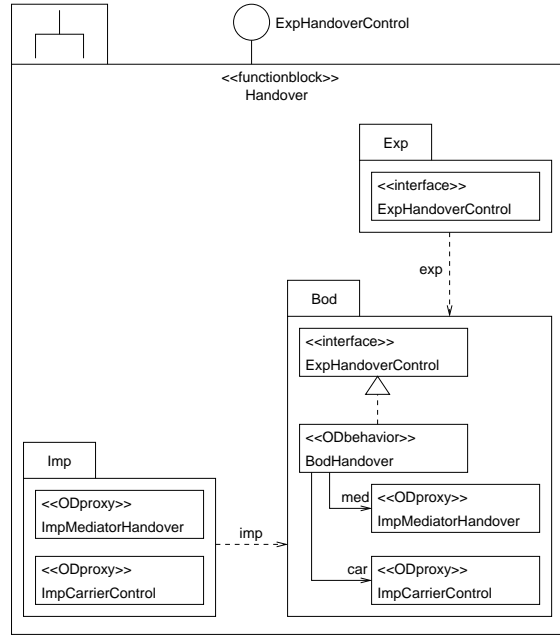
An abstract component is shown in Fig. 10. The import specification is connected to the body specification by an import connection  $imp: Imp \rightarrow Bod$ , where this connection should be a parameterization, i. e. it specifies, how the imported entities are used in the construction of the body. On the other hand, the export is related to the body by an export connection  $exp: Exp \rightarrow Bod$ , which is an abstraction from the details in the body, i. e. it specifies, how the abstract provisions in the export are realized by the body.



**Fig. 10.** Specification component

UML subsystems can be seen as an instantiation of this concept, where the operations offered by the interfaces of a subsystem together with its specification elements form the export interface, and the realization elements correspond to the body of the component. The UML does not demand explicit modeling of the import of a subsystem, but we have added this by redirecting dependencies to the environment through interfaces with the stereotype `<<ODproxy>>`, which have no realization in the subsystem itself but are instantiated by dependencies to the environment. The connections are then given by inclusions of the interface elements into the whole subsystem. The UML subsystem `Handover` in Fig. 5 can for example be interpreted as a component with the UML interface `ExpHandoverControl` in its export and the proxy interfaces `ImpMediatorHandover` and `ImpCarrierControl` in its import. The body then consists of all three interfaces together with the class `BodHandover`. This component structure is made explicit in Fig. 11

There are two approaches to the semantics of components. In [15] a model-theoretic approach is sketched, which can be seen as a generalized version of the algebraic module concept in [17]. In this approach, depicted in Fig. 12, the semantics of an import connection  $imp$  is given by a construction  $Constr_{imp}: Mod(Imp) \rightarrow Mod(Bod)$ , which builds models satisfying the body specification from models satisfying the import specification. An export connection  $exp$  is interpreted by a restriction  $Restr_{exp}: Mod(Bod) \rightarrow Mod(Exp)$ , which abstracts from the details of the realization in models of  $Bod$ , leading to models of  $Exp$ . Then, the model semantics  $ModSem(Comp): Mod(Imp) \rightarrow Mod(Exp)$  of a component  $Comp$  is just the composition  $ModSem(Comp) = Restr_{exp} \circ Constr_{imp}$ . Thus, the model semantics of the subsystem `Handover` is a function transforming models of the import interfaces into models of the export interface. A model semantics for

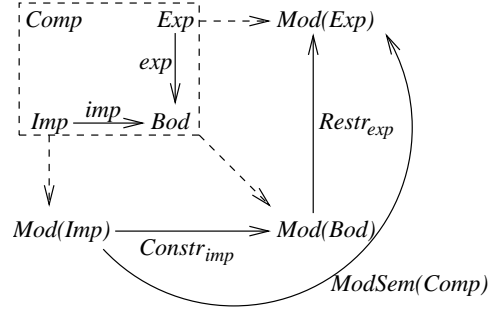


**Fig. 11.** Component Handover with explicit component structure

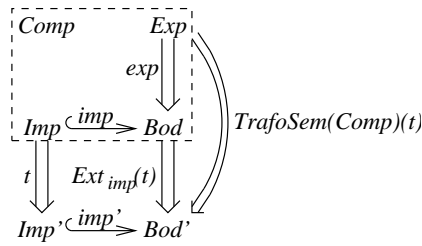
UML specifications together with constructions and restrictions is sketched in Sect. 4.

The second approach to component semantics, taken in [5], is based on transformations of specifications, where these transformations are supposed to somehow refine the specifications, and a class of inclusions of specifications. The transformation semantics  $Trafo(Spec)$  of a specification is then given by the class of all possible transformations  $t: Spec \Rightarrow Spec'$ . Transformations and inclusions have to be composable by a composition operator  $\circ$ . Moreover, the extension property has to be satisfied, i. e. for each inclusion  $i: Spec_1 \hookrightarrow Spec_2$  there has to be an extension function  $Ext_i: Trafo(Spec_1) \rightarrow Trafo(Spec_2)$ , such that for each transformation  $t: Spec_1 \Rightarrow Spec'_1$  with extension  $Ext_i(t): Spec_2 \Rightarrow Spec'_2$  there is an inclusion  $i': Spec'_1 \hookrightarrow Spec'_2$ . The semantics of a component, which now has to have an inclusion as import and a transformation as export connection, is then a function  $TrafoSem(Comp): Trafo(Exp) \rightarrow Trafo(Exp)$  with  $TrafoSem(Comp)(t) = Ext_{imp}(t) \circ exp$ , which means that a component can construct transformations of the export specification from transformations of the import specification. This situation is shown in Fig. 13.

A transformation based component concept for UML diagrams has already been examined in [18], where the transformations are inheritance relations between elements in an export package and elements in the corresponding body package and inclusions correspond to the import of the elements in an import package. Considering UML subsystems the transformations correspond to adding



**Fig. 12.** Model-theoretic semantics of component

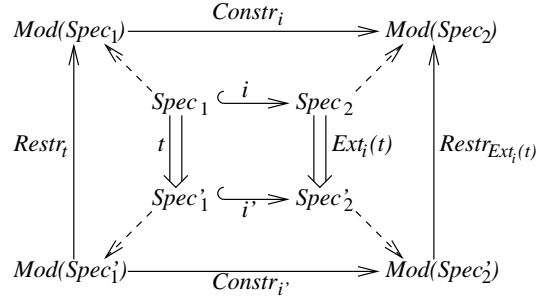


**Fig. 13.** Transformation semantics of component

the satisfying realization in the realization elements of the subsystem to the specification in the export interface. Especially implementing classes as for example `BodHandover` in Fig. 11 are added to UML interfaces like `ExpHandoverControl`. The import inclusions are in this case obviously the inclusion of the elements marked as `<<ODproxy>>` into the whole subsystem.

The relation between model and transformation semantics could be given by the requirement, that there has to be a construction  $Constr_i$  for each inclusion  $i$  and a restriction  $Restr_t$  for each transformation  $t$  and these have to be compatible with composition. This means, for  $i_1: Spec_1 \hookrightarrow Spec_2$  and  $i_2: Spec_2 \hookrightarrow Spec_3$  we have  $Constr_{i_2 \circ i_1} = Constr_{i_2} \circ Constr_{i_1}$  and for  $t_1: Spec \Rightarrow Spec'$  and  $t_2: Spec' \Rightarrow Spec''$  we have  $Restr_{t_2 \circ t_1} = Restr_{t_1} \circ Restr_{t_2}$ . Another interesting compatibility condition is compositionality of the model semantics with extension. This means for  $i: Spec_1 \hookrightarrow Spec_2$  and  $t: Spec_1 \Rightarrow Spec'_1$  with  $Ext_i(t): Spec_2 \Rightarrow Spec'_2$  and  $i': Spec'_1 \hookrightarrow Spec'_2$  we have  $Restr_{Ext_i(t)} \circ Constr_{i'} = Constr_i \circ Restr_t$  (i.e. the outer square in Fig. 14 commutes).

In other words, the inclusions and transformations of the transformation semantics approach, which could in general be interpreted arbitrarily, can be required to be sound w. r. t. the model semantics. After soundness has been proven for inclusions and transformations, the developer using the specification technique does not need to deal with the model semantics directly anymore, but can use the transformation semantics, which enables the refinement of specifications by syntactical transformations.



**Fig. 14.** Compatibility of model semantics with extension

In order to instantiate the requirements requested by the import specification, the composition of components yielding larger components is useful. In Fig. 15 the components  $Comp_1$  and  $Comp_2$  are composed via the connector  $conn: Imp_1 \rightarrow Exp_2$  leading to a new component  $Comp_3$ . Regarding the transformation semantics approach  $conn$  has to be a transformation, so that the extension  $Ext_{imp_1}(exp_2 \circ conn) = ce'$  exists due to the extension property. Regarding the model semantics, there should be a restriction  $Restr_{conn}: Mod(Exp_2) \rightarrow Mod(Imp_1)$ . Note, that compositionality of the model semantics can be deduced from the compatibility of transformation and model semantics:

$$\begin{aligned}
& ModSem(Comp_3) \\
&= Restr_{exp_3} \circ Constr_{imp_3} \\
&= Restr_{exp_1} \circ Restr_{Ext_{imp_1}(exp_2 \circ conn)} \circ Constr_{imp_1'} \circ Constr_{imp_2} \\
&= Restr_{exp_1} \circ Constr_{imp_1'} \circ Restr_{conn} \circ Restr_{exp_2} \circ Constr_{imp_2} \\
&= ModSem(Comp_1) \circ Restr_{conn} \circ ModSem(Comp_2)
\end{aligned}$$

The composition of components also leads to a composition operation for the instantiation to UML subsystems. With this composition architectures of UML subsystems with dependencies between their respective interfaces can be flattened by removing the redirections and proxy classes for satisfied imports and connecting the contents of the different subsystems directly. For example the architecture shown in Fig. 4 could be flattened to a single subsystem with no imports and just the interface `ExpHandoverControl` in the export as shown in Fig. 16.

In [19] the concepts of union and multiple interfaces for transformation components based on high-level replacement transformations, a generalization of graph transformations, have been additionally treated. These concepts are also a useful line of future work for this instantiation of the abstract component concept, because especially multiple import and export interfaces arise very naturally from the usage of UML subsystems.

Considering the notion of component in UML 2.0 introduced in the substructure specification in [2], it seems promising to also interpret these as instantiation of the abstract component concept. UML 2.0 components have provided and re-



quired interfaces and ports, that can be used as export and import specifications in the sense of the abstract component concept. Moreover, protocol state machines can be assigned to interfaces and ports, which allows to add constraints for the applicability of operations to the export and import specifications. Together with a model-theoretic semantics for (parts of) UML 2.0 similar to the one in the next section a formal notion of correctness could be derived for UML 2.0 components, which could enhance the reliability of specifications significantly, if practicable methods to proof this correctness are provided.

## 4 A Model-Theoretic Semantics for UML Subsystems

In this section we use object-oriented transformation systems (OOTS), which are variants of the transformation systems of Große-Rhode (see [8]), as a model-theoretic semantic domain for UML subsystems. The structure of OOTS and the specification by UML diagrams are presented in more detail in [16] in this volume.

An OOTS is given by a control transition graph, where the nodes are labeled with object configurations and the transitions are labeled with sets of actions executed during the transition. Object configurations contain the values of attributes for all existing objects of the system, where links between objects are considered as a special case of attributes. Actions can be calls and returns of methods, signal occurrences, and assignments of attribute values. Each OOTS is built over a static algebra  $St$  containing data type sorts with corresponding data functions. The entities of an OOTS (data type sorts, data functions, object sorts, attributes, signals, and methods) are declared in a data space signature  $DSig$ . The possible configurations and actions w. r. t. this data space signature then form the data space  $\mathbf{D}_{DSig}$ . Figure 17 shows an example OOTS, where we can also see that the actions are parameterized with parameters from the static part for data type sort parameters and from the configurations for object sort parameters. Method calls, signal occurrences and attribute assignments get an additional parameter as the first one. This parameter denotes the receiving object for a method call or signal or the object, whose attributes will be changed, for assignment actions.

Each specification has an associated signature, which induces the data space of OOTS models for that specification. For example the import specification  $Imp$  of Handover as shown in Fig. 11 has the associated data space signature  $ImpSig$  given in Fig. 18. Note that we also give abbreviated names (in square brackets), which will be used in the following figures for readability reasons. The configurations of the data space  $\mathbf{D}_{Imp\Sigma}$  contain sets of existing objects for the object sorts  $ImpCarrierControl$  and  $ImpMediatorHandover$ . The actions in the data space are the occurrences  $occ_{ICC.eL}$  and  $occ_{ICC.eU}$  of the signals and the call and return actions  $call_{IMH.sL}$  and  $ret_{IMH.sL}$  for the method `submitLoad` and  $call_{IMH.sU}$  and  $ret_{IMH.sU}$  for the method `submitUnload`. The models  $Mod(Imp)$  of the import specification are now all OOTS for the data space  $\mathbf{D}_{Imp\Sigma}$ , for which the statechart in Fig. 9 is satisfied, i. e. the signals `executeLoad` and `executeUnload`

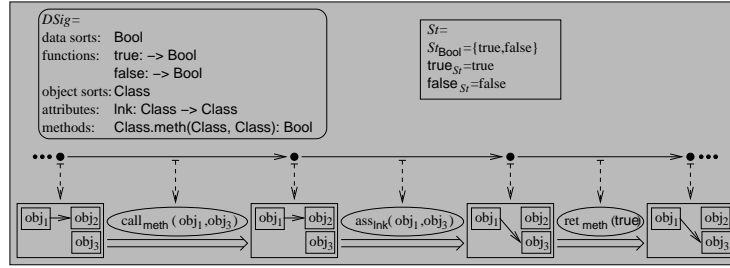


Fig. 17. Object-oriented transformation system

can be sent alternately. Since no statechart is given for `ImpMediatorHandover`, we assume that both methods are always callable. Because an OOTS for the import specification will in most cases be a restriction (see below) of a refined specification, it will usually contain transitions labeled with an empty set of actions, denoting that the events are not visible w.r.t.  $\text{Imp}\Sigma$ . A part of an example OOTS for the import of `Handover` is given in Fig. 18.

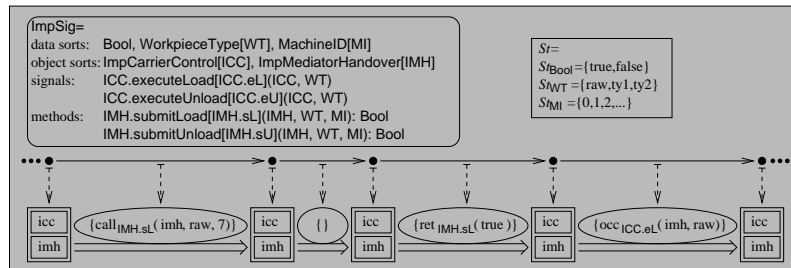
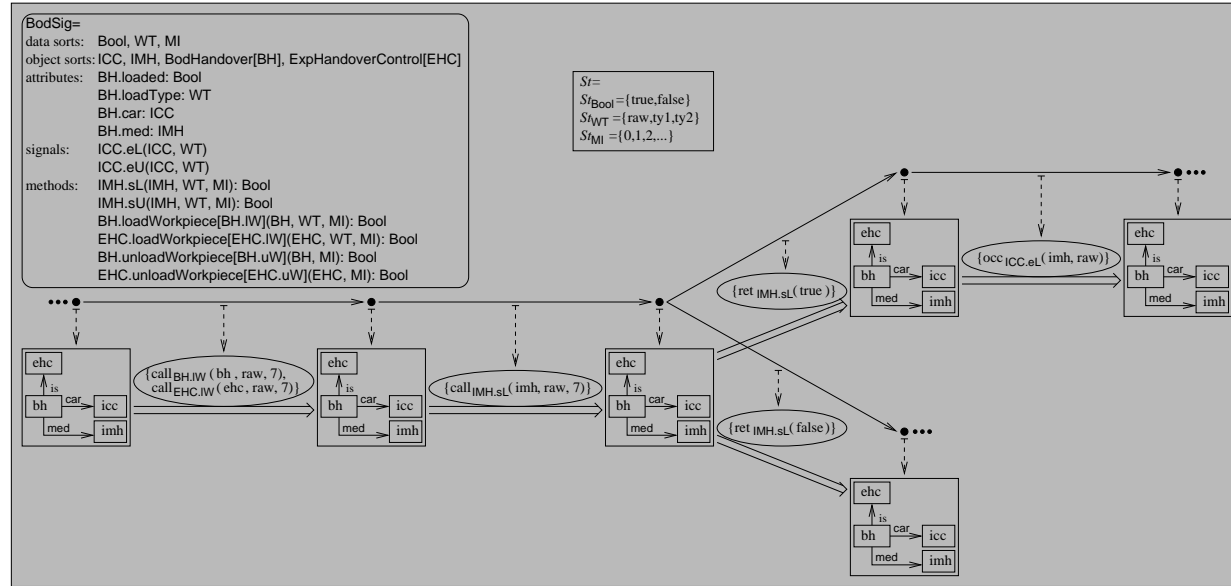


Fig. 18. OOTS for the import specification

For the body specification from Fig. 11 some entities are added to the data space signature shown in Fig. 19, namely object sorts for the class `BodHandover` and the interface `ExpHandoverControl`, the attributes of `BodHandover` and the methods available for `BodHandover` and `ExpHandoverControl`. Now we can construct a minimal model for the body specification, a part of which is shown in Fig. 19. This model is minimal in the sense that the internal structure of imported methods is omitted. This means they are only represented by their call actions and return actions as e.g. the method `submitLoad` in the example. For each invocation of an imported method it contains all return actions, which an instantiation of the import could possibly generate, since the body model has to be able to interact with arbitrary instantiations.

In Fig. 19 we can also see, how inheritance is represented in an OOTS: Since, the class `BodHandover` inherits from `ExpHandoverControl`, each object of `BodHandover` is also an object of `ExpHandoverControl`, e.g. the object `bh` in the

Fig. 19. Minimal OOTS for the body specification



object configurations is the object `ehc`. The call and return actions are included for both, the object and method of the implementing class and the object and method of the abstract class.

According to the model theoretic approach for the semantics of components presented in [15] and discussed in Sect. 3 the semantics  $ModSem(Comp)$  is given as composition  $ModSem(Comp) = Restr_{imp} \circ Constr_{exp}$ . The construction  $Constr_{imp}: Mod(Imp) \rightarrow Mod(Bod)$ , which is the semantics of the import inclusion `imp`, can now be obtained by synchronizing any given model in  $Mod(Imp)$  with this minimal model leading to a model in  $Mod(Bod)$ , which uses the imported model to realize the imported signals and methods. This synchronization is done by inserting the structure of imported methods from the import model into the minimal model. The synchronization of the OOTS part of the minimal model from Fig. 19 with the part of an import model from Fig. 18 is shown in Fig. 20, where the transition labeled with the empty action set, which represents the execution of the method `submitLoad`, is added to the minimal model of the body. The part of the minimal model dealing with a return value of `false` is not contained in the synchronization, because the import model returns `true` for this invocation.

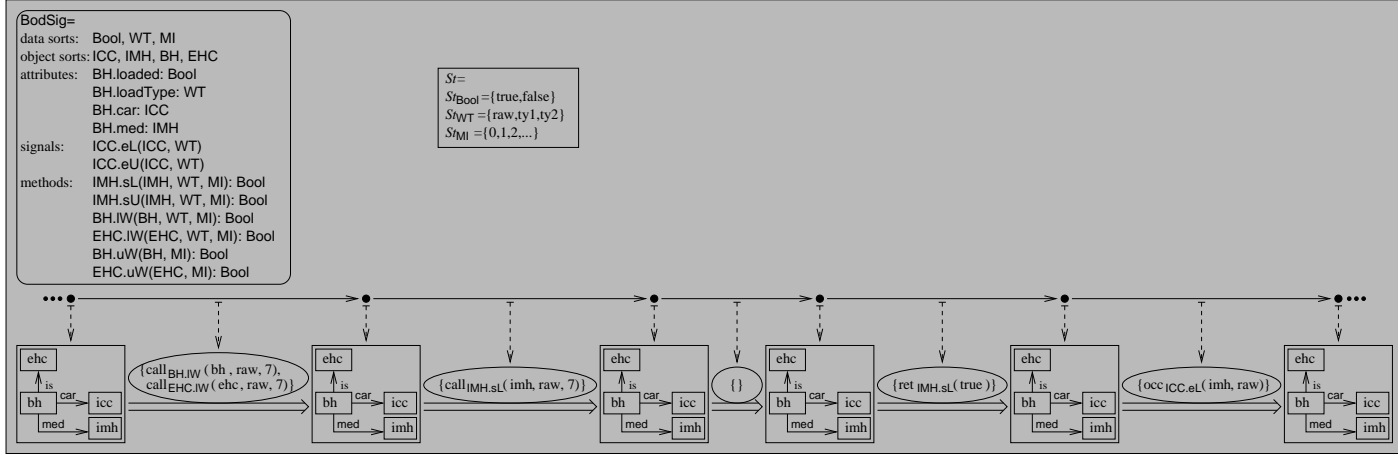
Finally, the restriction  $Restr_{exp}$ , which is the semantics of the export inclusion `exp`, hides all parts of models of `Bod`, except for the object sort of `ExpHandoverControl` and its associated methods. The restriction of the partial body OOTS from Fig. 20 is shown in Fig. 21.

## 5 Conclusion

Along the example of the specification of a holonic transport system it was shown, how the advantages of the component notion can be carried over to the development of embedded, object oriented systems. This was achieved by establishing a correspondence between the UML subsystem and the IEC function block notion. In order to further enhance the usability of this concept, as well as the complementary concept of viewpoint consistency in [16], more aspects of specific interest for production automation, e.g. real-time constraints, will be integrated into the model semantics and specification techniques based on, but not limited to, the UML will be examined w. r. t. these aspects.

In order to have available well defined composition operations and a formal semantics we sketched out how UML subsystems can be understood as components in an abstract and formal framework. This relation also has to be worked out in more detail to make available code generation, correctness notions, etc. for the modeling of components and function blocks with subsystems. Especially the relation between the different formal composition operations (horizontal composition, union, multiple interfaces) and the UML syntax has to be clarified.

Furthermore, the relations to the UML meta-model and components in the sense of UML 2.0 will be examined, in order to develop guide lines for tools supporting the concepts shown in this paper and the orthogonal concepts in [16]. More specifically, the component concept would induce a notion of correctness



**Fig. 20.** Constructed OOTS for the body specification

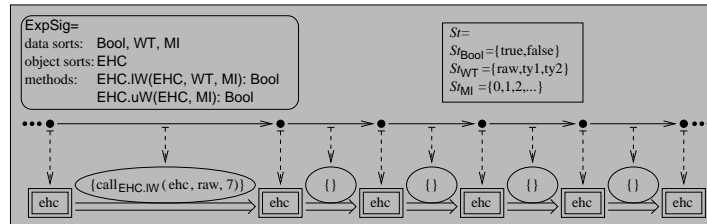


Fig. 21. Restricted OOTS for the export specification

of components together with administration guidelines for such components, which could e.g. minimize and observe the proof obligations for correctness and composition of components.

**Acknowledgements:** This work has been supported by the IOSIP project in the DFG priority program SoftSpez. We would like to thank Hartmut Ehrig and Martin Große-Rhode and the referees for their valuable comments on previous versions of the paper.

## References

1. Object Management Group: Unified Modeling Language – Version 1.5 (UML 1.5). (2003) Available from <http://www.omg.org/>.
2. Object Management Group: Unified Modeling Language – Version 2.0 (UML 2.0). (2004) Available from <http://www.omg.org/>.
3. Braatz, A., Klein, M., Ehrig, H., Westkämper, E.: Konzeption und Entwicklung eines UML-basierten Funktionsblockmodells für den objektorientierten Steuerungsentwurf. In: Entwicklung und Betrieb komplexer Automatisierungssysteme (EKA 2003), Institut für Regelungs- und Automatisierungstechnik, TU Braunschweig (2003)
4. Westkämper, E., Braatz, A.: Eine Methode zur objektorientierten Software-Spezifikation von dezentralen Automatisierungssystemen mit der Unified Modeling Language (UML). at – Automatisierungstechnik **5** (2001) 225–233
5. Ehrig, H., Orejas, F., Braatz, B., Klein, M., Piirainen, M.: A Generic Component Concept for System Modeling. In Kutsche, R.D., Weber, H., eds.: Fundamental Approaches to Software Engineering (FASE 2002). Number 2306 in Lecture Notes in Computer Science, Springer (2002) 33–48
6. Parnitzke, D.: On Formal Semantics of Object Systems with Data and Object Attributes. Forschungsbericht 2001/05, Fachbereich Informatik, TU Berlin (2001)
7. Tenzer, J.: A Formal Semantics of UML Class Diagrams based on Transformation Systems. Forschungsbericht 2001/09, Fachbereich Informatik, TU Berlin (2001)
8. Große-Rhode, M.: Semantic Integration of Heterogeneous Software Specifications. Monographs in Theoretical Computer Science. Springer (2004)
9. Klein, M., Braatz, B., Ehrig, H., Schröter, G., Bengel, M.: Anwendung softwaretechnischer Komponentenkonzepte auf die Produktionsautomatisierung. atp – Automatisierungstechnische Praxis (2004) To appear.
10. International Electrotechnical Commission: IEC/PAS 61499-1 – Function Blocks for Industrial-Process Measurement and Control Systems – Part 1: Architecture. (2000)

11. Braatz, A.: Entwicklung einer Methode zur objektorientierten Spezifikation von Steuerungen. PhD thesis, Universität Stuttgart (2004) submitted.
12. Braatz, A., Ritter, A.: Referenzfallstudie Produktionstechnik (PA) v2.0 (2001) Available from <http://tfs.cs.tu-berlin.de/~iosip/>.
13. Ehrig, H., Große-Rhode, M.: Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen. *Informatik Forschung und Entwicklung* **16** (2001) 110–117
14. Klein, M., Oezhan, M., Piirainen, M.: IOSIP Case Study Model Files (2002) Available from <http://tfs.cs.tu-berlin.de/~iosip/>.
15. Ehrig, H., Orejas, F.: A Generic Component Framework for Integrated Data Type and Process Modeling Techniques. Forschungsbericht 2001/12, Fachbereich Informatik, TU Berlin (2001)
16. Braatz, B., Klein, M., Schröter, G.: Semantical Integration of Object-Oriented Viewpoint Specification Techniques. In Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E., eds.: *Integration of Software Specification Techniques for Applications in Engineering*. Number 3147 in *Lecture Notes in Computer Science*. Springer (2004)
17. Ehrig, H., Mahr, B.: *Fundamentals of Algebraic Specification 2 – Module Specifications and Constraints*. Volume 21 of *Monographs on Theoretical Computer Science*. Springer (1990)
18. Piirainen, M.: Applications of a Generic Component Framework to a UML Case Study in Production Automation. Diploma thesis, TU Berlin (2003)
19. Ehrig, H., Orejas, F., Braatz, B., Klein, M., Piirainen, M.: A Component Framework for System Modeling Based on High-Level Replacement Systems. *Software and System Modeling* **3** (2004) 114–135