# Attribute Grouping: Emulating Metamodels without Instantiation


**Markus Baumeister**

*Lehrstuhl für Informatik V, RWTH Aachen*
*Ahornstr.55, 52056 Aachen, Germany*
*E–mail: baumeister@informatik.rwth-aachen.de   Fax: +49-241-8888321*



***ABSTRACT.*** *Attribute grouping is proposed as a representation of metamodels without an instantiation relationship between metaclasses and classes. This representation solves the problems which appeared while we tried to define a metamodel for chemical process modeling: implementability in conventional OO systems, placement of classes, and definition of instance–instance–attributes. It retains all the important features of metamodels which we need for data modeling except the ability to define class attributes and class methods. Besides providing an axiomatic definition of three different representations of metamodels, we also show how attribute grouping can easily be implemented in C++.*



***KEY WORDS:*** *Metamodeling, instantiation, specialization, object–oriented programming, chemical process design*


## 1 Introduction

The use of metamodels in modeling information systems has been growing over the last years. Many application areas like database integration (Jeusfeld and Johnen, 1995), multimedia object modeling (Klas and Schrefl, 1995), business process re–engineering (Nissen *et al.*, 1996), or method engineering (Hong *et al.*, 1995) profit from the ability of metamodels to describe a modeling domain in a highly abstract manner. We wanted to apply this principle to chemical process modeling, too, in a cooperation project at the RWTH Aachen between the Lehrstuhl für Prozeßtechnik (chemical process engineering (CPE)) and the Lehrstuhl für Informatik V (information systems). In the course of this we met some problems which, together with a possible solution, are described in the subsequent chapters.

Concrete models of chemical processes are used in the design of new and in the improvement and control of existing chemical plants respectively. Normally modeling is followed by simulating the model, which generates information on the behavior of the processes (according to the model). An overview over the current ways of modeling in CPE is presented in (Marquardt, 1996).

Goals for the design–environment of chemical process models, in development at the CPE–group in Aachen, are re–usability of existing model parts and support of inexperienced designers. To facilitate this, differences between specific model parts are abstracted away in model building blocks (like reactor, pipeline, evaporator or continuously stirred tank reactor with cooling jacket). These building blocks are arranged as classes in an extensible specialization hierarchy to simplify the modeling of similar building blocks and their retrieval for re–use.

This class–instance–architecture does not answer the question, though, how the abstract principles of modeling (i.e. the composition of a model out of devices, connections, interfaces, equations, etc. and their interdependencies) should be represented. Separating these principles from the model building blocks simplifies both the understanding of the modeling methodology and the implementation, since the methodology can be considered stable while building blocks may be defined and refined by the user.

Separating the modeling principles onto a higher abstraction level is called metamodeling(cf. (Nissen *et al.*, 1996)); Device, Connection, Equation, etc. are called metaclasses of which the building blocks are instances. A more detailed explanation of our view of metamodeling can be found in chapter two. In the third chapter we will present different representation methods for metamodels, which will be analyzed and compared in chapter four. The last chapter summarizes the paper.

## 2 Metamodeling

### *2.1 Application areas and systems*

Even though metaknowledge, metaformulas, metaprogramming, or metaphysics can be regarded as forms or applications of metamodels, we only reflect on metamodeling within the scope of class–based, object–oriented data models. As classes abstract instances within these models, metaclasses abstract classes. Thus classes are instances of metaclasses, metaclasses are instances of metametaclasses and so forth. Due to this, metamodels can be used to provide information about models (called *introspection* since metamodels are also models), to change the behavior of models/classes, i.e. provide an *adaptable* modeling *language*, and to extract an abstraction out of models (this will be called *model modeling* in the following).

Introspection and language adaption are the main uses of metamodels in those current programming languages providing them. Both CLOS–MOP (Kiczales *et al.*, 1991) and Smalltalk–80 (Goldberg and Robson, 1989) contain one metaclass level to allow applications the introspection of their schema and, to a greater or lesser extent, the change of the language. Attempts to extend two–level programming languages like C++ by a metalevel (Johnson and Palaniappan, 1993; Chiba and Masuda, 1993) normally also aim at these goals: (Johnson and Palaniappan, 1993) offers a specifiable amount of type information of classes at run–time, while (Chiba and Masuda, 1993) presents a simple metaobject protocol for C++ facilitating the change of method calling conventions.

Our main use for metamodels is model modeling; all application areas mentioned at the beginning of this paper fall into this category. One could claim, that there is no difference between model modeling and language adaption, since by defining a metamodel one just provides an adaptable language to specify the model in. But this language has no need to be adaptable in it's own language like CLOS or Smalltalk. On the other hand, we need a possibly unlimited number of metalevels instead of just one: Imagine after modeling a specific chemical process model (instances), its building blocks (classes) and the modeling principles (metaclasses) we want to compare the modeling principles of our chemical modeling language with the principles of another. We need a metametamodel abstracting from the specifics of the languages and showing differences and similarities. After that we want to compare the way we compared both languages with the comparison in another paper ... . This view of model modeling differs slightly from the IRDS–architecture (ISO, 1990), an metamodeling architecture suggested for information resource management, which provides only four object levels.

Our view of metamodeling is based on TELOS (Mylopoulos *et al.*, 1990), a deductive, object–based modeling language. The object model of TELOS differs from standard object models (like ODMG (Cattell, 1994)) in some points. It does contain metaclasses, even an unlimited number of levels of them. The problem of indefinite regress is solved by reflexive instantiation. All units of information (i.e. instances, classes, attributes, rules, etc.) are regarded as first class objects, enabling e.g. instantiation links and specialization links between attributes. Among the differences not influencing the capability to metamodel in principle are declarative rules (instead of methods), constraints and views, and the lack of encapsulation. There are several TELOS–based database systems, one of them *ConceptBase* (Jarke *et al.*, 1995), which we used for our work. A more thorough description of TELOS, metamodeling and it's history can be found in (Nissen *et al.*, 1996).

A similar, but fully object–oriented database system, VODAK, is presented in (Klas and Schrefl, 1995). (Klas and Schrefl, 1995) provide examples on how to use the metaclass–level of VODAK for both language extension (by defining different semantic relationships like "role–specialization") and abstract model modeling (by providing a metamodel consisting of "node" and "link" for hypertext structures).

All mentioned languages (i.e. CLOS, Smalltalk, TELOS, and VODAK) relate classes and metaclasses by an "instance_of" relationship. The problems caused by this instantiation link will be met again below.

### *2.2 Features and problems*

If one looks at our description of metamodeling in a more operational way, one can distinguish the following four modeling facilities (called '*functions*' from now on) of metaclasses. These functions shall be illustrated with the help of figure 1a, which contains a small part of the (meta)model of VeDa ((Marquardt *et al.*, 1993)), the modeling language used and developed by the CPE–group in Aachen. The four sectors of the rectangle correspond each to one level in the instantiation hierarchy, i.e. they contain from bottom to top: the concrete process model as instantiated by the user, the building blocks of process models as defined by knowledge engineers and users, the modeling methodology as researched by the knowledge engineers, and the definition of the language VeDa as defined by language and database developers.

The *functions* of metaclasses are:

**Collect classes:** The metaclasses (i.e. the objects of the modeling methodology level) collect classes according to their affiliation to basic principles of the chemical process model. E.g. `Reactor, Evaporator, Column` are instances of `Device`, `Pipeline` and `Wall` are instances of `Connection`. (*Function 1*)

**Group attributes:** At the same time as collecting classes, attributes such as `Product_out` are related to their corresponding attribute category (or meta–attribute); in this example `interfaces`. (*Function 2*)

**Define methods and constraints on instances:** An example for a constraint on instances (i.e. members of the model level) is `all_interfaces_eq_subdevice_interfaces`, a rule stating that in each instance of a class being instance of `CompositeDevice`, the interfaces of the device must be matched by the interfaces of it's subdevices. It should be mentioned that this function is not really provided by metaclasses but only

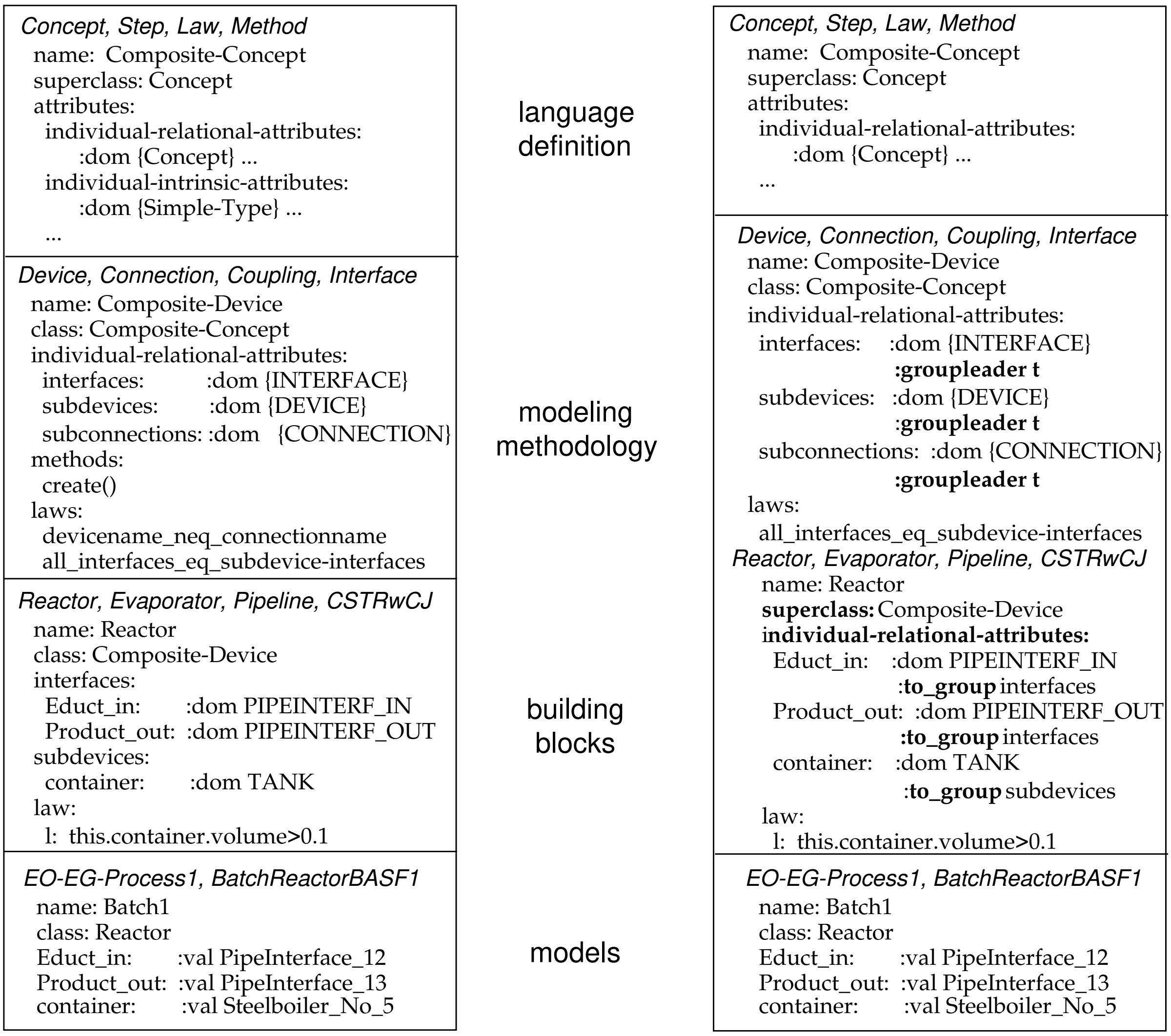


Figure 1: a) (Meta)model of a small part of VeDa; b) Metamodel–emulation by attribute–grouping

achieved by 'tricks' based on them like metarules (Jeusfeld, 1992) in *ConceptBase* or instance–instance–types (Klas and Schrefl, 1995) in VODAK. (*Function 3*)

**Define attributes, methods and constraints on classes:** Examples for class methods and class constraints are `create()`, for generating new device instances, and the somewhat artificial `deviceattrname_neq_connectionattrname`, prohibiting similar names of subdevice attributes and subconnection attributes. (*Function 4*)

These four functions are weighted differently in the three metamodel application areas mentioned above. *Introspection* heavily depends on the two grouping functions (function 1 and 2) of metaclasses, while *language adaption* places most of it's emphasis on defining behavior and attributes of classes and, to a lesser extend, instances. *Model modeling* possesses a more balanced need for all four functions but with a slight emphasis on the two grouping functions and a sub–average need for defining class attributes and methods.

Most modeling methodologies in chemical process modeling do not use metaclasses for model modeling or only implicitly during the definition of their modeling language (Baumeister, 1996). In the light of the additional modeling facilities provided by metamodels this can only be explained by the problems arising due to the use of metamodels:

1. Lack of (commercial) databases to make the model persistent
2. Incompatibility of metamodeling and standard programming languages like Eiffel or C++
3. Level–jumping attributes, i.e. attributes of the instances which should be defined in the metaclasses (e.g. `OID` for the metaclass class)
4. Difficulties to fit some classes in either the class or the metaclass level

Two annotations to the database and programming language problems seem appropriate: First, the lack of databases is not just a subset of the programming language problem, since it would be sufficient for some applications to view a two–level area of a metamodeling–capable database.

Second, the incompatibility of metamodeling and standard two–level programming languages is caused by the problems of mapping at least three object levels onto only two. Up to now there were two ways of action:

- Use classes of the programming language to represent metaclasses: This way one either has to do without the class level (i.e. `Batch1` was a direct instance of `Device`) causing a loss of classifyability of objects, or one has to model instances *and classes* in the instances level of the programming language, meaning ultimately a self–implemented object system.
- Use language classes to represent model *classes*: Since here metaclasses have no existence of their own, the information contained in them has to be preserved somehow else. The mechanisms of the class level, especially inheritance, are not well suited for this, since they cannot step out of the abstraction level of the class level. Also changes in the metaclasses may cause problems due to redundantly represented information.

So both current remedies for the incompatibility of metamodeling and standard programming languages loose information and/or need a considerable amount of additional code.

To solve these problems we searched for an alternative for metamodeling providing equivalent functionality. This is described in the next section.

## 3 Alternative representations for metamodels

Normally metamodels are represented by instantiation relationships between metaclasses/ meta–attributes and classes/attributes (as can be seen in figure 2a). But if one looks a little closer at this way of representation, one notices that the double instantiation relation between metaclasses and classes, and between classes and objects is responsible for many of the problems mentioned above: It is incompatible with standard two–level object models, and it causes the above problem number three by not being transitive. The solutions proposed in the following will therefore replace the upper instantiation relation by specialization or other relationships. To obtain a basis for comparisons, we'll first formalize 'normal' metamodeling.

### *3.1 Metaclasses*

Using instantiation relationships to couple metaclasses and classes can be formalized as follows – according to (Jeusfeld, 1992), though in a different notation. We assume here that attributes are first–class objects. Since defining a complete axiom system for object models is not the aim of our work, we concentrate on the axioms necessary for metamodel representation.

*Let $\mathcal{O}$ be the set of all individual objects, i.e. instances, classes, metaclasses, metametaclasses and so on, and let $\mathcal{I}$ be the set of all identifiers. Both sets shall be finite (without loss of generality). Let $\mathcal{A}$ be the set of all intensional attributes (i.e. $\mathcal{A} \subset \mathcal{O} \times \mathcal{I} \times \mathcal{O}$). Furthermore we define the relation $\longrightarrow \subset \mathcal{O} \times \mathcal{A} \times \mathcal{O}$ with $c \xrightarrow{a} d$ meaning that there exists an attribute $a$ pointing from $c$ to $d$. Name $\subset (\mathcal{O} \cup \mathcal{A}) \times \mathcal{I}$ is the relation between an object and it's identifier. Finally Attr $\subset \mathcal{O} \times \mathcal{I} \times \mathcal{O}$ is the relation of all attributenames between two objects ($\mathcal{A} \subset$ Attr).*

**A 1** $\forall\, c, d, mc, md \in \mathcal{O}, \forall\, a, ma \in \mathcal{A} : c \xrightarrow{a} d,\ mc \xrightarrow{ma} md,\ a \textit{ instance of } ma$
$\Rightarrow c \textit{ instance of } mc,\ d \textit{ instance of } md$
(i.e. if attribute $a$ is an instance of meta–attribute $ma$ there must be the same relation between the source and target objects of the two attributes.)

**A 2** $\forall\, c, d, mc, md \in \mathcal{O}, \forall\, a, ma \in \mathcal{A}, \forall\, l \in \mathcal{I}$ :
$c \xrightarrow{a} d,\ mc \xrightarrow{ma} md,\ a \textit{ instance of } ma,\ \textit{Name}(ma, l) \Rightarrow \textit{Attr}(c, l, d)$
(i.e. if $a$ is an instance of $ma$, then the relation between source and target object of $a$ is additionally known under the name of $ma$.)

Axiom A2 provides the grouping of the class attributes by metaclass attributes (function 2), while axiom A1 restricts the possible groupings. The collecting of classes is achieved by the instantiation links emanating to the common metaclass of the classes. Class attributes, class methods and class constraints are defined similar to instance attributes, methods and constraints by classes. As already mentioned, with instantiation–based metaclasses it requires additional mechanisms like metarules or instance–instance–types to define attributes, constraint and methods of instances in the metaclasses.

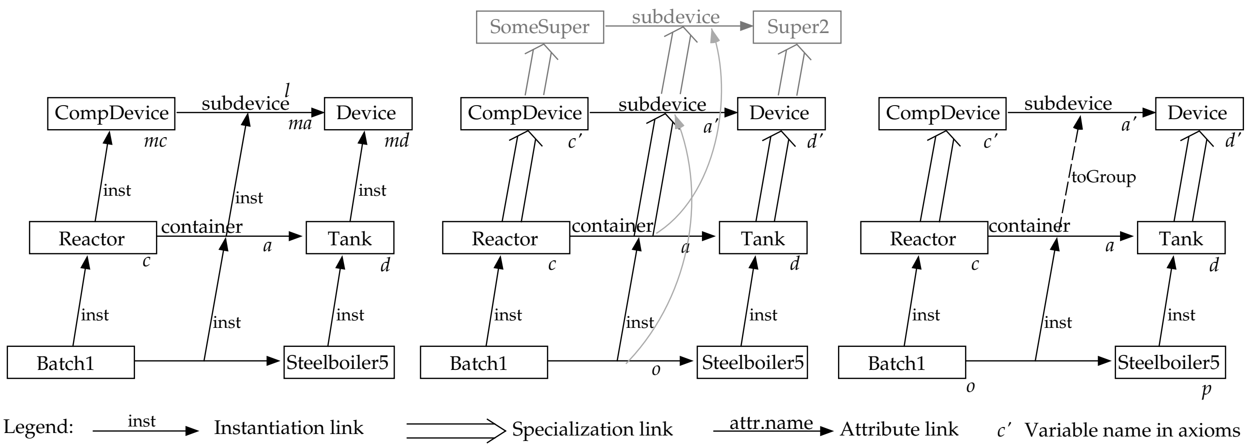


Figure 2: Metamodeling and it's emulations: a) 'real' metamodeling; b) multiple attribute–specialization; c) attribute–grouping

### *3.2 Multiple attribute–specialization*

In this alternative representation we replace the instantiation relationship between classes and metaclasses, and between attributes and meta–attributes by a specialization relationship. The instantiation relationship between classes and instances is of course not affected. In this way all the levels of the metaclass hierarchy above the class level are mapped onto the class level.

The imitation of metamodeling by multiple attribute–specialization (i.e. specialization relationships between attributes in subclasses and superclasses independently from their identifiers[1]) can be formalized as follows:

*Let $\mathcal{K}$ be the set of all classes, $\mathcal{I}$ the set of all identifiers and $\mathcal{O}$ the set of all instances. Again all sets shall be finite. Let $\mathcal{A}$ be the set of all attributes (i.e. $\mathcal{A} \subset \mathcal{K} \times \mathcal{I} \times \mathcal{K}$).*
*Again $\longrightarrow \subset \mathcal{K} \times \mathcal{A} \times \mathcal{K}$ with $c \xrightarrow{a} d$ meaning that there exists an attribute $a$ pointing from $c$ to $d$. Name $\subset (\mathcal{K} \cup \mathcal{A}) \times \mathcal{I}$ is the relation between an object and it's identifier. Finally Attr $\subset \mathcal{O} \times \mathcal{I} \times \mathcal{O}$ is the relation of the attribute names between two instances.*

**B 1** $\forall c, c', d, d' \in \mathcal{K}, \forall a, a' \in \mathcal{A} : c \xrightarrow{a} d,\ c' \xrightarrow{a'} d',\ a\ isA\ a' \Rightarrow c\ isA\ c',\ d\ isA\ d'$
(i.e. the source object and the target object of an attribute must be specialized analogously to the attribute.)

**B 2** $\forall c, c' \in \mathcal{K} \cup \mathcal{A}, \forall o \in \mathcal{O} : c\ isA\ c', o\ instance\ of\ c \Rightarrow o\ instance\ of\ c'$
(i.e. instances of a subclass are also instances of the superclass. This axiom is also necessary in the other axiom systems to describe the mechanism of specialization, but it is not involved in metamodel representation there.)

**B 3** $\forall c, c', d, d' \in \mathcal{K}, \forall a, a' \in \mathcal{A}, \forall l \in \mathcal{I}$ :
$$c \xrightarrow{a} d,\ c\ isA\ c', c' \xrightarrow{a'} d',\ Name(a, l),\ Name(a', l) \Rightarrow a\ isA\ a'$$
(This axiom maps the 'natural' attribute–specialization onto the multiple one.)

Axiom B2 groups attributes, since it results in $Attr(o, l, o'), Name(a, l), Name(a', l')$ and $a\ isA\ a'$ implying $Attr(o, l', o')$. This grouping is restricted by B1. But in contrast to the previous representation this alternative allows some relations not possible there (the grey arrows in figure 2b):

- A class $c''$, interposed between $c$ and $c'$ in the specialization hierarchy, specializes attribute $a'$ of $c'$. But instead of being derived from this specialization $a$ directly specializes $a'$. Formally:
  $\exists c'' : c\ isA\ c''\ isA\ c', c'' \xrightarrow{a'} d'', Name(a'', l), Name(a', l), d''\ isA\ d'$.
- 'Virtual' meta–attributes (i.e. multiply specialized attributes) can have instances while 'real' meta–attributes can't.

Furthermore without 'first class attributes' multiple attribute–specialization becomes problematic, since it is no longer easily integratable with natural attribute–specialization.

### *3.3 Attribute–grouping*

To achieve an alternative representation of metamodels which more closely reproduces the behavior of the instantiation–based one as specified in section 3.1 we introduce a new relation 'toGroup' and a set called 'groupleader'; for a first impression have a look at figure 2c.

Since it is one of our goals to reach a representation compatible with standard OO databases and programming languages, the following axiom system will not contain attributes as first class objects but assumes they only exist in conjunction with classes/instances[2].

[1]Please note that specializing attributes in any way is not allowed in strongly typed OO languages wanting to retain object substitutability. Therefore multiple attribute–specialization is only a partial step forward in the case of the implementation/database problems, since many databases are strongly typed.

[2]Figure 2c does not comply with this, since it is intended to show the similarities and differences of the methods in the same graphical notation.

*Let $\mathcal{K}$ be the set of all classes, $\mathcal{I}$ the set of all identifiers and $\mathcal{O}$ the set of all instances. As always all these sets shall be finite. Furthermore c.a denotes the non–inherited attribute a of the class c and Attr $\subset$ $(\mathcal{K} \times \mathcal{I} \times \mathcal{K}) \cup (\mathcal{O} \times \mathcal{I} \times \mathcal{O})$ is the relationship between the attributes belonging to a class/object and their target classes/-objects (on the classlevel again only non-inherited attributes are looked at[3]):*

**C 1** $\forall\, c, c', d, d' \in \mathcal{K}, \forall\, a, a' \in \mathcal{I} : Attr(c, a, d),\ Attr(c', a', d'),\ c.a\ toGroup\ c'.a'$
$\Rightarrow c\ isA\ c',\ d\ isA\ d',\ c'.a' \in groupleader,\ \neg(\exists c'' \in \mathcal{K} : c\ isA\ c''\ isA\ c' : c''.a')$
(i.e. if there exists a 'toGroup' relationship between two attributes, the source and target classes of the grouped attribute must be specializations of those of the grouping one. Moreover this grouping attribute has to be a member of the 'groupleader' set, which therefore contains all (virtual) meta–attributes. Finally, no class specializing the grouping attribute may be placed on the specialization path between the two classes. )

**C 2** $\forall\, c, c' \in \mathcal{K}, \forall\, a \in \mathcal{I}, \forall\, o, p \in \mathcal{O} :$
$c.a\ toGroup\ c'.a',\ o\ instance\ of\ c,\ Attr(o, a, p) \Rightarrow Attr(o, a', p)$
(i.e. in instances the grouped attributes are also known by the name of the grouping attribute.)

**C 3** $\forall\, c \in \mathcal{K}, \forall\, a \in \mathcal{I} : c'.a' \in groupleader \Rightarrow \neg\exists\, o \in \mathcal{O} : o\ instance\ of\ c',\ o.a'$
(i.e. grouping attributes may not appear in instances.)

In the same way as in the previous axiom systems the C2–axiom groups the attributes, i.e. it ensures that grouped attributes are also visible by the name of their groupleader and that constraints and methods of the groupleader affect them. Axiom C1 restricts the possible groupings avoiding the interposed class problem multiple specialization had, while axiom C3 disallows instantiating virtual meta–attributes, thereby resolving the second discrepancy of multiple specialization. Classes are collected by specialization links; instance attributes, constraints and methods are gained without special tricks (since there is no longer an object level between metamodels and instances); only class attributes, class methods, etc. cannot — similar to multiple attribute specialization — be defined by the new virtual metaclasses.

The right part of figure 1 shows the representation of VeDa's (meta)model with only two instance–class pairs instead of three, since the modeling methodology level and the building block level are now represented within one level using attribute–grouping. Since it was never intended to represent the language definition within the database, there remains only one instance–class pair to be implemented in the database.
Important differences to the left part of the figure are printed in bold.

## 4 Analysis

The previous chapter presented some alternative representations for metamodels and contained some hints on the quality of these replacements. We'll now expand on this.

Substituting specialization links for the instantiation links between classes and metaclasses (section 3.2) allows us to retain three of the four metaclass functions without actually needing metaclasses. Indeed it even offers a little "more" functionality than possible with instantiation. Since this "more" means a loss of semantic we switch to a new relation (section 3.3)

[3]It is sufficient to consider non–inherited attributes, since a combination of axioms B2 and B3 defines inheritance here as well.

for which the axioms are not determined by an externally fixed semantic. Therefore we can choose the axioms in a way removing the glitches of multiple specialization. But though this leads to a close approximation of the axioms of 'real' metamodeling, we are still not able to realize it's fourth function (class methods, constraints, and attributes; cf. p.3).

On the other hand our mapping of metamodels to two–level object systems without first class attributes enables us to easily implement attribute–grouping in OO programming languages like C++ and consequently also in most OO databases. Figure 3 shows the translation, which could be done automatically, of the classes `CompDevice` and `Reactor` from figure 1b into C++. The groupleader attributes (i.e. the grouping attributes) are implemented as

```
class CompDevice:  public Device
{
    virtual set<Interface> getinterfaces() { return(empty_set); }
    virtual set<Device> getsubdevices() { return(empty_set); }
    ...
}

class Reactor :  public CompDevice
{   Pipeline_In *Educt_In;
    Pipeline_Out *Product_Out;

    virtual set<Interface> getinterfaces()
        { return(union(CompDevice::getinterfaces(), Educt_In, Product_Out)); }
    ...
}
```

Figure 3: Attribute–grouping in C++

virtual functions which collect the grouped attributes when their corresponding groupleader attribute is accessed. While axioms C2 and C3 are preserved by the structure of the above program, the checking of axiom C1 has to be done by a precompiler, which would have to convert a more user–friendly notation to the above one. Comparing this implementation with the possibilities of representing metamodels in two–level systems mentioned on page 5 we can see, that this implementation strategy, while preserving most metamodel information, only needs a minor amount of additional code by reusing mechanisms of the programming language.

An implementation of our metamodel representation in extendible languages like CLOS or VODAK seems also feasible, due to the inherent language adaption capabilities even without a preprocessor, providing these languages with the possibility to represent more than the three object levels they currently provide.

Up to now we only mentioned the problems of representing metamodels in two–level object systems. The two remaining problems — level-jumping attributes and arrangement of classes — are solved by both multiple attribute–specialization and attribute–grouping:

- Since there is only one instantiation level left between virtual meta–attributes and instance attributes, the problem of level–jumping attributes, methods, etc. vanishes. This is, though, the reason for the inability to define class attributes (metamodel function 4).

- Because there are no longer two orthogonal hierarchies into which classes are placed, but only the specialization hierarchy, an appropriate position for a class is easier to locate, especially since class and metaclass properties can be combined in one class.

Therefore we can see that attribute–grouping, while retaining very closely three of the four metamodel functions, solves the problems caused by metamodels.

## 5 Conclusion and Outlook

We showed that using metamodels to abstract modeling knowledge (in chemical process engineering and elsewhere) differs from other applications of metamodels like language adaption. We further pointed out problems occurring with this model modeling use, among others the difficulties of implementing the resulting data model in conventional OO programming languages and databases.

Starting from the axiomized properties of instantiation–based metaclasses we proposed a new way to represent metamodels, which is especially easy to implement in two–leveled OO programming languages. Due to this, metamodels of data can be realized within programs without having to wait for a commercial programming language capable of real metamodeling.

As it has been pointed out, attribute–grouping does not allow the definition of class attributes, class methods, and class constraints. Therefore it can't provide facilities like language adaption similar to the metaobject protocol of CLOS. Further research could try to solve this by replacing the specialization relationships between classes with self–defined relations. It needs to be considered, though, if this is really worthwhile, since it will probably prevent an implementation as easy as shown in chapter 4, since it might require re-implementing parts of the object model of the programming language.

**Acknowledgment**
The author would like to thank M. Jarke, M. Jeusfeld, M. Staudt, S. Sklorz and K. Pohl for suggestions, comments and discussions.

## References

M. Baumeister. *Ein objektorientiertes Datenmodell zur Beschreibung chemischer Prozesse.* In Informatik und Technik, Forschungsbeiträge des Aachener Graduiertenkollegs. Graduiertenkolleg 'Informatik und Technik', RWTH Aachen, Germany, 1996.

R. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, 1994. ISBN 1-55860-302-6.

S. Chiba and T. Masuda. *Designing an Extensible Distributed Language with a Meta-Level Architecture.* In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 483–502, Kaiserslautern, Germany. Springer-Verlag, July 1993.

A. Goldberg and D. Robson. *Smalltalk-80: The Language.* Addison-Wesley, Reading, MA, 1989.

S. Hong, S. Brinkkemper and F. Harmsen. *Object–Oriented Method Components for Situation–Specific IS Development.* In S. Ram and M. Jarke, editors, *Proceedings of the Fifth Anual Workshop on Information Technologies & Systems (WITS'95)*, pages 164–173, Aachen, Germany. RWTH Aachen, Fachgruppe Informatik, 1995.

ISO. *Information technology — Information Resource Dictionary System (IRDS) framework.* ISO/IEC International Standard 10027, first edition, 1990.

M. Jarke, R. Gallersdörfer, M. Jeusfeld and S. Eherer. *ConceptBase: A deductive object base for meta data management.* Journal of Intelligent Information Systems, 4(2):167–192, March 1995.

M. Jeusfeld and U. Johnen. *An executable meta model for re-engineering of database schemas.* Intl. Journal on Cooperative Information Systems, 4(2&3):237–258, 1995.

M. A. Jeusfeld. *Änderungskontrolle in deduktiven Objektbanken.* Infix–Verlag, St. Augustin, Deutschland, 1992.

R. Johnson and M. Palaniappan. *MetaFlex: A Flexible Metaclass Generator.* In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 503–528, Kaiserslautern, Germany. Springer-Verlag, July 1993.

G. Kiczales, J. des Rivieres and D. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

W. Klas and M. Schrefl. *Metaclasses and Their Application.* LNCS 943. Springer, Berlin, Heidelberg, 1995.

W. Marquardt. *Trends in Computer–Aided Process Modeling.* Computers chemical Engineering, 20(6/7):591–609, 1996.

W. Marquardt, A. Gerstlauer and E. Gilles. *Modeling and Representation of complex Objects: A chemical Engineering Perpective.* In Proceedings of the Conference on Intelligent Engineering Applications/Artificial Intelligence in Engineering, Edinburgh. June 1993.

J. Mylopoulos, A. Borgida, M. Jarke and M. Koubarakis. *Telos: Representing Knowledge About Information Systems.* ACM Transactions on Information Systems, 8(4):325–362, 1990.

H. Nissen, M. Jeusfeld, M. Jarke, G. Zemanek and H. Huber. *Managing Multiple Requirements Perspectives with Metamodels.* IEEE Software, 13(2), March 1996.