# Dimensions and Dichotomy in Metamodeling

Robert Geisler*, Marcus Klar** and Claudia Pons***

* Technische Universität Berlin, Institut für Kommunikations- und Softwaretechnik
Franklinstr. 28/29, 10587 Berlin, Germany,
E-Mail: geislerr@cs.tu-berlin.de
** Fraunhofer Institut für Software- und Systemtechnik ISST
Mollstraße 1, 10178 Berlin, Germany,
E-Mail: Marcus.Klar@isst.fhg.de
*** Universidad Nacional de La Plata. LIFIA
Casilla de Correo 11. CP: 1900 La Plata, Argentina,
E-Mail: cpons@sol.info.unlp.edu.ar

Metamodeling is playing an increasingly important role in object-oriented software engineering. However, most approaches use metamodels in a very pragmatic way and important conceptual questions are left open.

In this paper, an object-oriented metamodeling methodology based on a formal metalanguage is introduced. The methodology allows for the description of all relevant properties of a metamodel, i. e. abstract syntax, static and dynamic semantics. Different kinds of instantiation relations are identified and a dichotomy for the classification of metaentities is developed. The reflection of the instantiation relations by the metalanguage is shown.

## 1 Introduction

Metamodeling is used as a general technique for integrating and defining models from different domains. Common aspects of these different views can be identified and shared. The metamodeling technique consequently can be applied in quite different application domains, especially for standardization purposes. Metamodels should therefore be rigorously defined as well as being intuitive and well-structured. Many general methodological aspects of modeling are also valid for metamodeling, however, further issues need to be considered in detail.

In this paper we present a metamodeling methodology based on a formal metalanguage. This methodology allows for the description of all relevant aspects of the entities of the metamodel. We are particularly interested in the description of dynamic semantics because this aspect has been neglected in common metamodeling approaches. The metamodel level, as well as the model level, are described. We provide a dichotomy for the classification of metaentities into intensional and extensional entities. Two kinds of instantiation relations are identified: inter-level instantiation and intra-level instantiation. The compatibility of these relations is proved.

The examination of behavior of intensional as well as extensional enities within a single approach may lead to an integration of process aspects of software engineering with the used modeling techniques.

### 1.1 Metamodels for the Definition of Multiple View Languages

A widely used and generally accepted technique in modern software engineering is the combination of different models (or views) for the description of software systems. The primary benefit of this approach is the modeling of related aspects (like structure or behavior). For this principle, called *Separation of Concern*, different specialized techniques, mostly of diagrammatic nature, have been developed. The use of different models clarifies different important aspects of the system, but it has to be taken into consideration that these models are dependent on each other and are semantically overlapping. Therefore, it is necessary to state how these models are related. The different views on a system have to be semantically compatible and there are several constraints between them.

Metamodeling is a very promising technique for the definition of multiple view languages like the *Unified Modeling Language* [UML97]. Using a metamodel, it is possible to determine how these models constitute the whole system. We would like to point out that while we are using an object-oriented approach to metamodeling, we are in no way restricted to the integration of object-oriented modeling techniques. Within our ongoing work, we have applied this approach to object-oriented modeling techniques [MK98], as well as to the integration of specification techniques [Gei98].

### 1.2 Outline

In Section 2, we sketch the commonly accepted four level approach to metamodeling that serves as the basis for our methodology. We introduce the dichotomy of intensional and extensional metaentities and discuss the general structuring mechanisms used in our methodology. A running example is used in Section 3 in order to demonstrate the application of the methodology. Key-concepts of metamodeling, such as *instantiation*, *metaclasses*, *abstract syntax*, and *static/dynamic semantics* are explained and exemplified. The crucial *Intensional/Extensional Dichotomy* is the subject of Section 4. We explain the different kinds of instantiation relations in detail and then show compatibility properties of the metalanguage with respect to these instantiation relations, e. g. that instantiation is reflected over different levels. Finally, in Section 5, we evaluate our approach using some common criteria for the evaluation of metamodels. We summarize the benefits of our approach and compare it to other existing approaches.

## 2 The Metamodeling Technique

We present a methodology for the definition of metamodels. The methodology is based on the four-level approach, see Section 2.1. The presented approach allows not only for the description of structural relations between the entities of a metamodel, but also for a formal definition of structural constraints and dynamic behavior of the entities. The entities of the metamodel are classified and the different kinds of instantiation relations are developed in a systematic way.

### 2.1 The four-level Approach

A metamodel is a model for the information that can be expressed during (software) modeling. Basically, a metamodel is a model of models. It consists of entities defining the model elements and thereby the modeling language. The main purpose of a metamodel is to relate these model elements. The different levels of abstraction are illustrated in Fig. 1 [Ode95].

On the data- and process-level, the entities are run-time objects, i.e. instances of classes

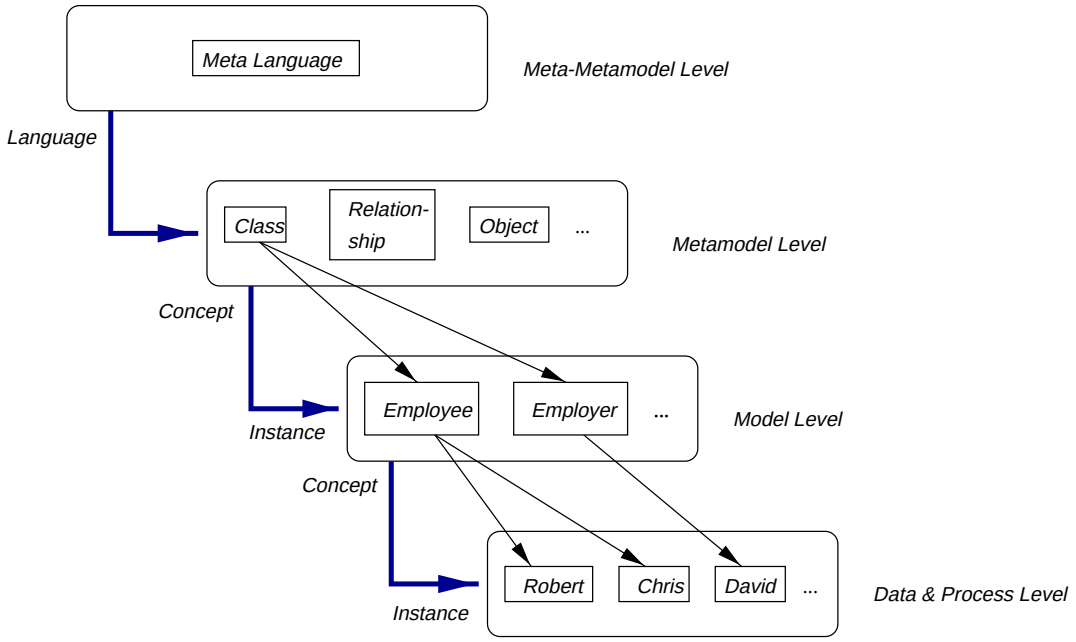



Figure 1: Different Levels of Abstraction

and processes running on a concrete system. On the level of models, we have different models describing the underlying physical system, e.g. *Employee* and *Employer* are classes of a structure diagram. The next abstraction, the metamodel level, describes the model in which, for example, the entities are *classes* and *objects*. Following the example given in Fig. 1, we consider *Employer* as an instance of *Class* of the metamodel. In order to express these concepts, we need a further level, defining the used language for the metamodel. This level is called meta-metamodel level.

The integration of different models using the metamodel is depicted in Fig. 2.[1] First, the elements of each model are interpreted in the metamodel. This interpretation, depicted as dashed lines, is usually done implicitly by the definition of the modeling language itself. Within the metamodel the modeling entities themselves, as well as the relationships between these entities are described, e.g. an *Association* consists of *AssociationEnd*s. Furthermore, entities of different models might be also related. Consider for example the association between *Class* and *Behavior* which establishes the relation between structural (*Class*) and behavioral elements (*StateMachine*). This exemplifies also the need for further constraints in order to exclude that a *StateMachine* is also related to another *StateMachine*. Generally, such constraints are very important in order to state the relationship between different entities of the metamodel.

According to [Atk97], we distinguish two metamodeling approaches. The *loose approach* permits that an instance of a metaentity (e.g. metaentity Object) occurs on the same level as its template (e.g. metaentity Class). In a *strict approach* a template is located on a higher level than its instance. In the following, we will concentrate on loose metamodeling.

It is also possible to consider more than four levels. A crucial question is how to terminate

[1] This example is taken from the UML.

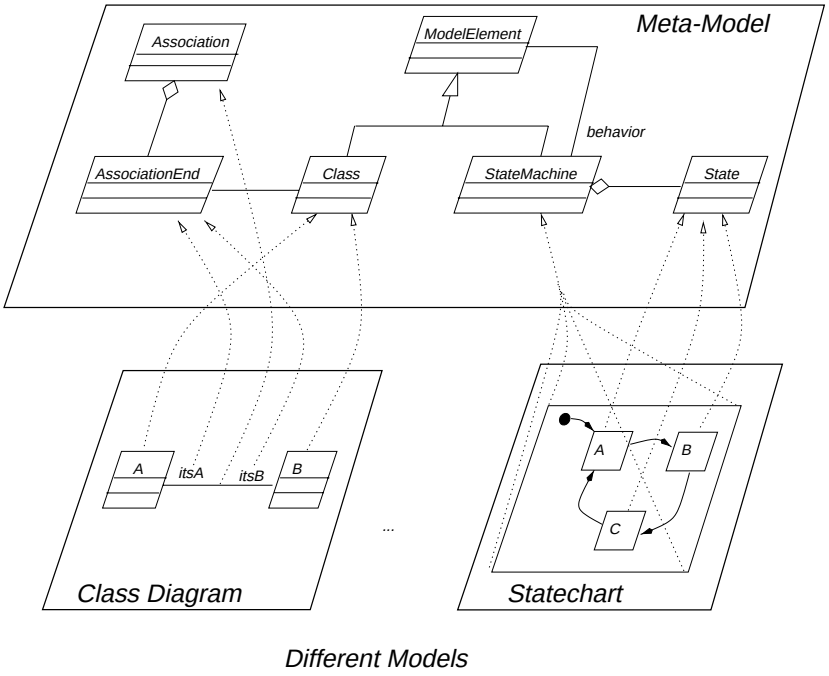



Figure 2: Using the Metamodel for the Integration of Different Models

the hierarchy. One technique is that the top-level model is an instance of itself. This solution gives rise to self-referencing problems. To avoid this, we propose the use of an independently-defined metalanguage. Overall, our approach is not only applicable for the definition of a metamodel but also to higher levels, e.g. the meta-metamodel level.

## 2.2 Intensional and Extensional Entities

An important contribution of this paper is the distinction between intensional and extensional metaentities. Intensional entities have a counterpart in the concrete syntax of the modeling language, such as *Class* or *Association*. In contrast, extensional entities, such as *Object* or *Link*, are used to store necessary run-time information. Extensional entities are usually closely coupled to an intensional entity in a way that the structure of the extensional entity is described by an intensional entity – for example, the structure of an *Object* is given by a *Class*. One of the major advantages of metamodels is the ability to define the relation of intensional and extensional entities within one level. In this way, the "semantics" of an intensional entity can be explained as the set of all corresponding extensional entities fulfilling the requirements. A remarkable feature of metamodeling (loose approach) is that the semantics of a metaentity can be defined within its own level.

## 2.3 Structuring Metamodels

A major focus of our work is the attempt to develop well-structured metamodels. For this purpose we use structuring techniques that are well-known from software engineering. First of all, we decompose a metamodel in different metaclasses. Within a metaclass, we are able to express all aspects relevant to a metaentity:

- Abstract Syntax: An abstract description of the entities that form a model of the respective language.
- Static Semantics (context conditions, constraints) : Well-formedness conditions between the syntactic entities, such as absence of circular inheritance.
- Dynamic Semantics (denotation): The (operational) behavior of the entities of the specification, such as I/O, reaction to stimuli, effect of executing an operation, etc.

The description of dynamic semantics in a metamodel, in particular, is not treated precisely in other approaches. The dimensions of metamodeling are depicted in Figure 3. Here, the concepts of intensional and extensional entities and its relation to abstract syntax, static and dynamic semantics is summarized.

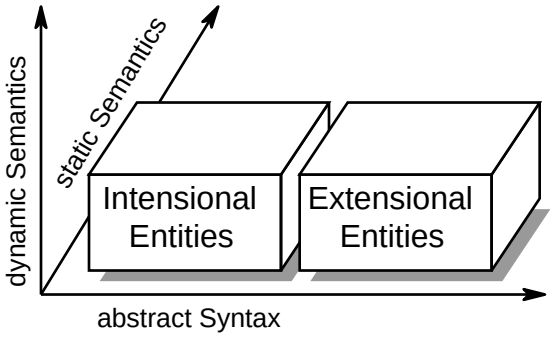



Figure 3: The Dimensions of Metamodeling

A metaentity has an internal state that is protected by object-oriented encapsulation principles. This ensures that each metaentity has its own meaning, as independent as possible from other metaentities. We call this principle, *localization principle* (see [Öve98]). In classical approaches for the definition of semantics of languages, e.g. [Mey88], this localization is not ensured. The advantages are twofold: first, we achieve a better understandability and readability. Second, and maybe more important, we achieve an increased flexibility with respect to extensions and changes. This has proved very important for the development of metamodels.

In general, a metaclass might be related to other metaclasses by an association relationship. An extensional entity is connected to its intensional entity by a special instantiation association. The division between intensional and extensional entities partitions the metamodel in two parts that are only connected by these special associations. In Section 4.2 we formulate guidelines concerning these associations. The generalization-specialization relationship is important for defining abstract metaclasses as interfaces for metaentities. These interfaces have to be fulfilled by the specialized metaentities. In Section 3, we illustrate the structuring techniques as well as the methodological aspects of our approach.

In Figure 4, the different roles of abstract syntax, static and dynamic semantics for intensional and extensional objects are summarized. The most remarkable difference is observable in the dynamic semantics. While dynamic semantics on the extensional level means run-time behavior, dynamic semantics on intensional level describes system evolution in the development process.

| | Intensional | Extensional |
|---|---|---|
| Abstract Syntax | specification in the modeling language(s) | state of a program at run-time |
| Static Semantics | well-formed specification | possible/consistent system states of a program |
| Dynamic Semantics | evolution of the specification during the modeling process | dynamic behavior of the program at run-time |

Figure 4: Intensional vs. Extensional Entities

### 2.4 The Metalanguage

It is commonly accepted that modeling languages require a formal semantics in order to be unambiguous. Since we use a metamodel for the definition of modeling languages, the metalanguage necessarily must have a formal semantics. According to 2.3, the metalanguage should support the following concepts:

- Objects: encapsulation of an object's internal state. An object defines an entity that may only be accessed in a controlled way.
- Object Identity: the notion of a persistent identity for an object. Therefore, object identities should be reflected in the semantics of the metalanguage.
- Object Behavior: the possibility of objects evolving in a pre-defined way. According to the *localization principle*, we propose the use of method definitions for the definition of object behavior.
- Classes: the ability to describe the common aspects of objects and encapsulate them in a class structure. Objects should then be instances of a type corresponding to the class. The features of an object should be accessible via methods.
- Compositionality (associations): the ability to declare a structural connection between classes so that a class can access services of the associated class.
- (multiple) Inheritance: defining a class as an extension of one or more existing classes.
- Polymorphism: the ability to define operations that act upon several distinct classes.
- Constraints: the ability to describe constraints between objects or pre- and post-conditions of object operations in some kind of predicate logic.

There are several formal languages fulfilling the above requirements of the metalanguage. In the following, we have chosen Object-Z [DRS94] as the formal metalanguage. Object-Z fulfills all requirements stated above and provides additional concepts (e.g. parameterization) that can be used in a sensible way. Furthermore, the structure of an Object-Z specification can be illustrated by a class diagram preserving the classical representation of metamodels.

## 3 Illustrating the Technique

In this section, we show how our approach can be used for the definition of modeling languages. Using a running example, we discuss concepts employed in our work.

### 3.1 The Running Example

We demonstrate our methodology by defining a metamodel for the integration of a simple non-deterministic automaton with a toy programming language. In Figure 5, the entities of the metamodel are shown using the UML notation. We first give a brief informal explanation of the semantic entities. In the following, we will demonstrate our method by describing the entities of the metamodel in our framework.

#### 3.1.1 Automata

An *Automaton* consists of a set of states, an initial state and a set of transitions. A *Transition* has a source and a target state. It is labeled with a condition - a boolean expression - and an action - a statement. Executing a transition causes the action to be executed (method *Execute*).

The semantics of an automaton is given by the associated class *Configuration*. During run-time, an automaton resides in exactly one state (*actualState*). The dynamic semantics of an automaton is given by the method *ExecuteStep* that chooses an enabled transition (i. e. the transition's source state is the actual state and its condition is evaluated true) and executes it.

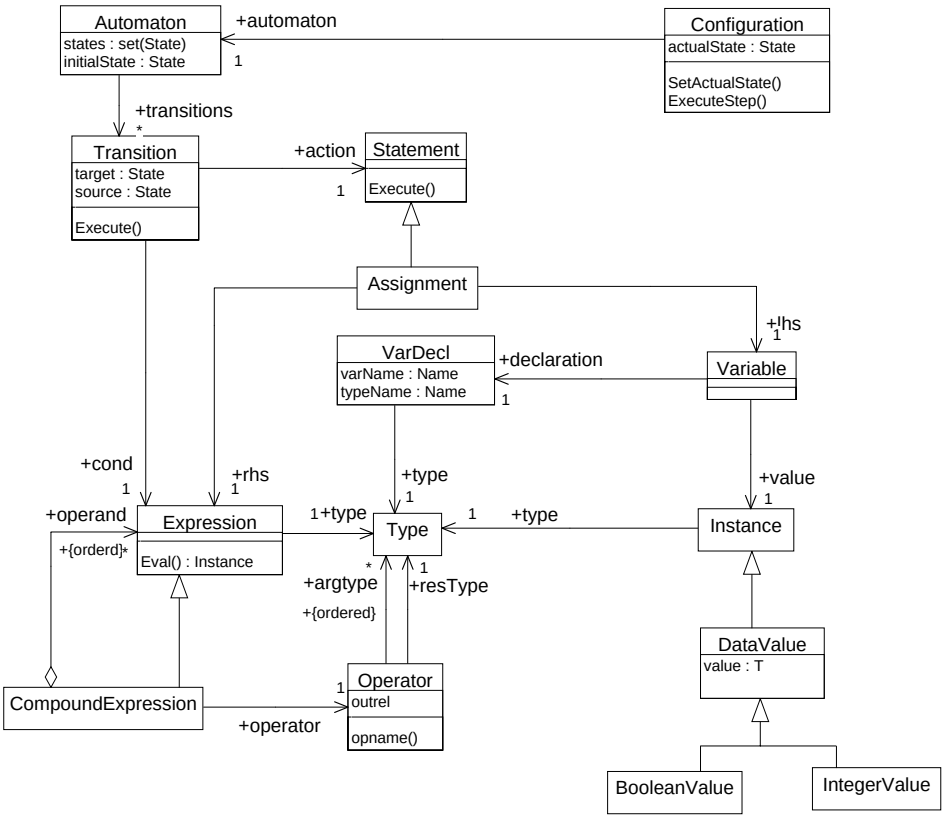



Figure 5: The Running Example

### 3.1.2 Programming Language

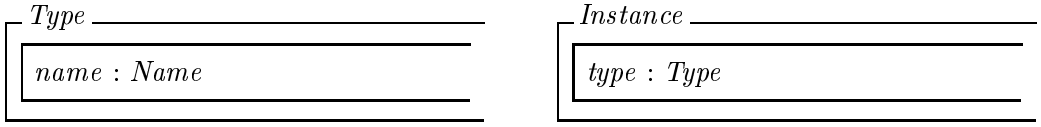
Our programming language basically consists of variable declarations (*VarDecl*), *Type*s, *Statement*s and *Expression*s. A *VarDecl* introduces a name and a type for a variable. A *Type* has a name – further details of *Type* are omitted. A *Statement*'s dynamic semantics is defined by the method *Execute* describing the effect of executing the statement. The only statement considered here is the *Assignment* statement, consisting of a *Variable* and an *Expression* that is to be bound to the variable. An *Expression* has a type and it can be evaluated using the method *Eval* that is returning an *Instance*. The only kind of expression we have in our example is a *CompoundExpression*, consisting of an *Operator* and a list of *operand*s.[2] An *Operator* has a signature (*argtype*, *restype*) and a semantics, determined by the attribute *outrel*, specifying how operands are mapped to the result.

A *Variable* has a *declaration* and is bound to an *Instance* at run-time. An *Instance* has a *type*. In our example, the only instances covered are *DataValue*s, carrying a *value*. We distinguish *BooleanValue*s and *IntegerValue*s.

## 3.2 Different Instantiation Relations

We can now begin with the definition of meta entities in the metalanguage Object-Z. A good introduction to Object-Z can be found in [DRS94].

There are different instantiation relations. In our example, we would like to state that an *Instance* knows its *Type*[3]. This is given by the association *type* in the class diagram or by the attribute *type* of the class *Instance*.




A completely different relationship is the instantiation of metamodel entities. For example, we would like to define the built-in types *int* and *bool*:

$$\begin{array}{|l} int, bool : Type \\ \hline int.name = Integer \land bool.name = Boolean \end{array}$$

Note that *Integer* and *Boolean* are predefined names.

With the above definition, we have crossed the border from metamodel level to model level (compare Figure 1) by *instantiating* a concept (*Type*) from the metamodel level. This shows that the instantiation relation between the different levels is mapped naturally to the instantiation mechanism provided by the metalanguage Object-Z.

## 3.3 Structuring of the Metamodel

The metalanguage provides several structuring mechanisms (e. g. parameterization, inheritance), which are also very useful for structuring the metamodel.

[2] Atomic expressions are those consisting of a constant operator and an empty operand.

[3] Here, we do not consider the internal structure of *Type*.

*DataValue*[*T*]

*Instance*

$value : T$

$\forall e_1, e_2 : DataValue[T] \bullet e_1 = e_2 \Leftrightarrow e_1.value = e_2.value$

The metaclass *DataValue*[*T*] inherits properties of *Instance*, i.e. a *DataValue* is an *Instance*. We define *DataValue* as a generic class over some internal datatype *T*. *DataValue*s of a programming language, in contrast to *Object*s that have a unique identity, share the characteristic of extensional equality.

We would like to introduce *BooleanValue*s and *IntegerValue*s. The instances of the classes *BooleanValue* and *IntegerValue* constitute the carrier sets for the built-in types *int* and *bool*. We use the internal data types ($\mathbb{B}$ and $\mathbb{Z}$) provided by the metalanguage for this purpose.

*BooleanValue*

*DataValue*[$\mathbb{B}$]

$type = bool$

*IntegerValue*

*DataValue*[$\mathbb{Z}$]

$type = int$

By the inherited constraint of extensional equality, it is assured that there are exactly two instances of *BooleanValue*, namely *tt* and *ff*.

$tt, ff : BooleanValue$

$tt.value \wedge \neg\, ff.value$

### 3.4 Abstract Metaclasses

An *abstract metaclass* is a metaclass that cannot be instantiated. It is only possible to instantiate subclasses of an abstract metaclass. Abstract metaclasses are used for the definition of interfaces within the metamodel.

*Expression*

$type : Type$

$\#Expression = 0$

$Eval \mathrel{\widehat=} [\, val! : \downarrow Instance \,]$

*Expression* is an abstract metaclass as required by the constraint $\#Expression = 0$. It provides an interface that has to be realized by all expressions, namely that every expression has a type and can be evaluated. This is expressed by the attribute *type* and the definition of the "virtual" method *Eval* that results in an *Instance*. The declaration $val! : \downarrow Instance$ means that the variable *val* is polymorphic, e.g. it is of type *Instance* or of any class type derived from *Instance* by inheritance.

### 3.5 Abstract Responsibilities of Metaclasses

One of the main purposes of metamodeling is to provide a complete and abstract description of the entities. We propose using structuring techniques for this purpose. Therefore, we

propose defining a general metaclass *Operator* and instantiating this class in order to obtain the pre-defined operators.

**Operator**

$$
\begin{array}{l}
argType : \mathrm{seq}\ Type \\
resType : Type \\
outRel : (\mathrm{seq}{\downarrow}Instance) \nrightarrow ({\downarrow}Instance) \\
\hline
\forall s : \mathrm{dom}\ outRel \bullet \#argType = \#s \\
\forall i : 1 \mathinner{..} \#argType;\ s : \mathrm{dom}\ outRel \bullet argType(i) = (s(i)).type \\
\forall i : \mathrm{ran}\ outRel \bullet resType = i.type
\end{array}
$$

**Eval**

$$
\begin{array}{l}
args? : \mathrm{seq}{\downarrow}Instance \\
inst! : {\downarrow}Instance \\
\hline
args? \in \mathrm{dom}\ outRel \\
inst! = outRel(args?)
\end{array}
$$

The signature of an *Operator* is defined by a sequence of argument types *argType* and a result type *resType*. For example the *lessOperator* (see below) takes two *int* expressions and results in a *bool* expression. The meaning of an *Operator* is defined by a partial function *outRel*. The constraints in the class *Operator* state that an *Operator* can only be applied to *Instance*s of its argument types. The method *Eval* takes a sequence of instances and delivers the instance that is yielded by applying the *outrel* relation to them.

$$
\begin{array}{l}
lessOperator : Operator \\
\hline
lessOperator.argType = \langle int, int \rangle \\
lessOperator.resType = bool \\
\forall i1, i2 : IntegerValue;\ b : BooleanValue \bullet \\
\quad (b = lessOperator.outRel(\langle i1, i2 \rangle) \land \\
\qquad b.value \Leftrightarrow (i1.value < i2.value))
\end{array}
$$

$$
\begin{array}{l}
plusOperator : Operator \\
\hline
plusOperator.argType = \langle int, int \rangle \\
plusOperator.resType = int \\
\forall i, i1, i2 : {\downarrow}IntegerValue \bullet \\
\quad (i = plusOperator.outRel(\langle i1, i2 \rangle) \land \\
\qquad i.value = (i1.value + i1.value))
\end{array}
$$

Based on *Operator*, we can now define a *CompoundExpression*, which is an *Expression* that consists of an *Operator* applied to its operands. The abstract *Eval*-operation has to be implemented: the operands are evaluated and the operator is applied to the results.

*CompoundExpression*

*Expression*

$operand : \mathrm{seq}\downarrow Expression$
$operator : Operator$

$\#(operator.argType) = \#operand$
$type = operator.resType$

$Eval \mathrel{\widehat=} [\, inst! : \downarrow Instance \,] \bullet$
$([\, args : \mathrm{seq}\downarrow Instance \,] \bullet (\fatsemi_9\ i : 1 \mathinner{..} \#operand \bullet$
$[\, e : \downarrow Expression;\ inst : \downarrow Instance \mid e = operand(i) \land inst = args(i) \,] \bullet$
$e.Eval[inst/inst!])$
$\fatsemi_9 operator.Eval[args?/args])$

### 3.6 Intensional and Extensional Entities

The difference between intensional and extensional entities is demonstrated by the following example:

A variable declaration is an intensional object, i.e. variable declarations are part of the modeling language. A variable declaration's state remains constant during computation time. The corresponding extensional entity is a *Variable*. It is linked to an intensional object, its declaration. It is storing run-time information over the variable, i.e. the value the variable currently holds. The dynamic semantics of variables is given by the method *Attach* that allows the changing of the value of a variable. It is typical for extensional objects that they provide methods with $\Delta$ lists that serve to alter their state.

*VarDecl*

$varName : Name$
$typeName : Name$
$\Delta$
$type : Type$

$type.name = typeName$

*Variable*

*Expression*

$declaration : VarDecl$
$value : \downarrow Instance$
$\Delta$
$type : Type$

$type = declaration.type = value.type$

*Attach*

$\Delta(value)$
$val? : \downarrow Instance$

$value' = val?$

$Eval \mathrel{\widehat=} [\, inst! : \downarrow Instance \mid inst! = value \,]$

### 3.7 Integration of different Modeling Languages

Now consider the class *Automaton* from Figure 5. Its transitions are labeled with conditions and actions:

*Transition*

$source, target : State$
$cond : \downarrow Expression$
$action : \downarrow Statement$

$cond.type = bool$

$Execute \mathrel{\widehat=} action.Execute$

A *Transition* consists of a source and a target state and is labelled with a boolean expression, denoting whether it can be executed or not. Its dynamic semantics is given by the operation *Execute*, executing its *action* – an arbitrary statement.

*Automaton*

$states : \mathbb{F}\ State$
$transitions : \mathbb{F}\ Transition$
$initialState : State$

$\forall\, t : transitions \bullet$
$\quad \{t.source, t.target\} \subseteq states$
$initialState \in states$

*Configuration*

$automaton : Automaton$
$actualState : State$

$actualState \in automaton.states$

*INIT*

$actualState = automaton.initialState$

*SetActualState*

$\Delta(actualState)$
$newState? : State$

$actualState' = newState?$

$ExecuteStep \mathrel{\widehat=} \ldots$

An *Automaton* represents an intensional object, i.e. the states, transitions and its initial state must be described in an appropriate modeling language. A *Configuration* is an extensional object that is linked to an intensional object, its *automaton*. It consists of an attribute *actualState*, which holds the state the automata is residing in. Its dynamic semantics is given by the methods *SetActualState* and *ExecuteStep*, which describe the behavior of the automaton.

### 3.8 Modifying the Model

The distinction between intensional and extensional entities becomes apparent if one considers the behavior of the metaentities. While the dynamic semantics of extensional entities defines the system dynamics, the dynamic semantics of intensional entities describes how the structure of the system might evolve. This feature is especially useful for describing the consistent modification of the model in the sense of system evolution.

Let us consider the following simple example of model modification: the renaming of a type.

*TypeRen*

*Type*

*Rename*

$\Delta(name)$

$newName? : Name$

---

$name' = newName?$

Using the operation *Rename*, it is possible to change the name of the type. The problem is that the model becomes inconsistent, since further model elements refer to the modified type by its old name. It is therefore necessary to change further modeling elements.

In our example, we have to rename the type name of the affected variable declarations:

*VarDeclRen*

*VarDecl*

*Rename*

$\Delta(typeName)$

$newName? : Name$

---

$typeName' = newName?$

A consistent type renaming renames the type itself, as well as all references to it:

*TypeRen2*

*Type*

$Rename \mathrel{\widehat=} [\, \Delta(name)\ newName? : Name \mid name' = newName? \,] \bullet$
$\quad {}^{\circ}_{9}\ v : VarDeclRen \mid v.typeName = name \bullet v.Rename$

Again, we take advantage of the reference semantics, seeing as we do not need to change further modeling elements, like *Variable* or *Expression*. Due to the persistent object identity, the changes are local. Since our model identifies the same concepts that are used in different models, the side-effects are minimal.

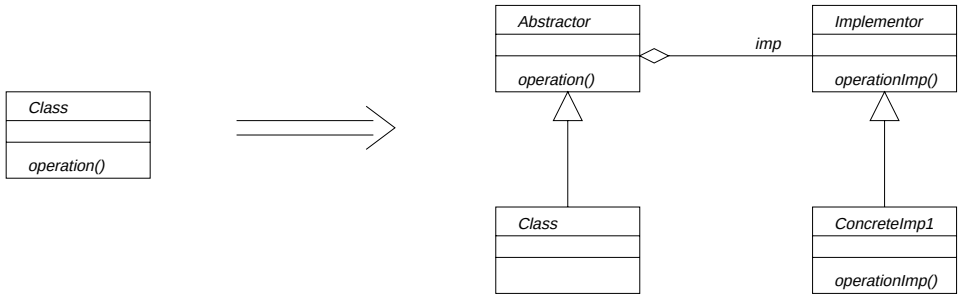



Figure 6: Introducing the Bridge Pattern

A more complex situation is the introduction of a design pattern in a software design. Consider, for example, the introduction of the bridge pattern depicted in Figure 6 (see [GHJV95] for a discussion of the pattern).

From a software engineering point of view, it is very important to consider how design patterns could be introduced within existing software designs. We believe that our metamodel based approach could serve as a foundation for the definition of such modifications of the system.

# 4 The Intensional/Extensional Dichotomy

The terms *Intension* and *Extension* are defined in [CN93] as follows:

> A term [an element of a proposition] may be viewed in two ways, either as a class of objects (which may have only one member), or as a set of attributes or characteristics which determine the objects. The first phase or aspect is called the denotation or extension of the term, while the second is called the connotation or intension. The extension of the term 'philosopher' is 'Socrates', 'Plato', 'Thales', and the like; its intension is 'lover of wisdom', 'intelligent', and so on.

In our metamodel, we distinguish between objects that have a counterpart in the syntax (intensional objects) and those that are of purely semantical nature, i.e. that carry run-time information (extensional objects). Extensional objects are linked via a special instantiation association with intensional objects. In our example, a *Variable* is an instantiation of a *VariableDeclaration*, and a *Configuration* is an instantiation of an *Automaton*. The role of updating methods is different for these kinds of objects. Updating (changing the state of) an intensional object means manipulating the model, whereas updating the state of an extensional object contributes to the dynamic semantics of one particular model.

## 4.1 Different Instantiation Relations

We provide two different kinds of instantiation relations in our metamodel approach. On one hand, we have special instantiation associations between the entities of one level of the metamodel. Such a relation connects an intensional object with an extensional object. Note that this is not characterized by a special syntactic construct of the metalanguage. Nevertheless, it is important to identify these associations properly. We call this kind of instantiation *intra-level instantiation*. Second, we have the instantiation mechanism of the metalanguage (i.e. *lessOperator* : *Operator*). This instantiation, called *inter-level instantiation*, causes a transition between levels. It represents the instantiation arrow between metamodel level and model level, see Figure 1. This instantiation preserves the characteristics of extensionality and intensionality, i.e. the instantiation of an intensional object on the metamodel level is an intensional object on the model level.

In Figure 7, the different instantiation relations are depicted. There are several views on a semantic entity:

- It can be viewed as an extension of some intensional object from its upper level.
- It can be instantiated by some object on its lower level, thus being viewed as an intensional object.
- Within its own level, a semantic object is either intensional, if it is directly reflected in the syntax of the modeling language, or it is an extension of an intensional object, representing information about the history of a computation.

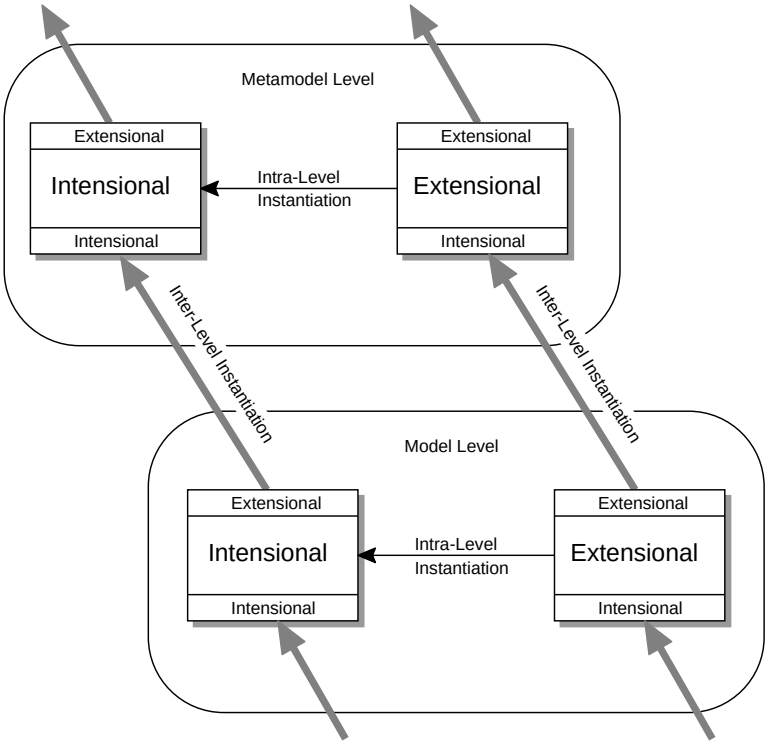



Figure 7: Instantiation Relations

## 4.2 Implications of the Dichotomy

In the following, we formulate some guidelines for the construction and evaluation of metamodels. We do not consider these guidelines obligatory. However, non-conformity of the metamodel with these guidelines should be carefully justified.

- *Classification of metaentities:*
  The entities of the metamodel should be divided into intensional and extensional entities. The intra-level instantiations (*instance_of* associations) should be distinguished. Normally, their cardinality should be $1 : n$ from intensional to extensional. Every intensional and every extensional entity should participate in, at the most, one intra-level instantiation. In our running example (see Fig. 5), the entities on the left side are intensional entities while the entities on the right are extensional.

- *Unambiguity of intra-level instantiation:*
  An extensional entity is connected to exactly one intensional entity by the *instance_of* association. In statically typed languages, the instantiated association (link), once established, remains constant, i. e. an object cannot change its type.

- *Preservation of dichotomy by inheritance:*
  The partition in intensional and extensional entities is preserved by the inheritance relationship. There exists no (concrete) metaentity that is super-type of an extensional and an intensional entity. Furthermore, no intensional entity is a super-type of an extensional entity and vice versa.

- *Reflection of associations between intensional entities via intra-level instantiation:* If two intensional entities are connected by an association, then this association is reflected by a corresponding association on the extensional level such that the traversal along the associations is compatible, i.e. the diagram commutes in a certain sense. This property is important since the *instance_of* relationship is reflected on the model-level by the inter-level instantiation.

### 4.3 Properties of the Metalanguage

We will now state some properties of the metalanguage that are useful for the understanding of the intensional/extensional dichotomy. In [GKP98], we have defined the semantic entities and functions formally. In this paper, we give just a short overview:

The semantics of a class is the set of all possible valuations that might be assigned to an object of the class. It is defined by the function *classSem* that assigns each class declaration the set of all possible bindings. The semantics of an instance declaration of the form $c : C$ is defined by the function *objSem* that assigns to each instance declaration an object (reference). The value of an object is defined by the function *objectMap* assigning the valuation for each object. The function *type* delivers for an instance declaration its type and represents the inter-level instantiation. The functions $asso_{CD}$, $asso_{CS}$, $asso_{OD}$ and *deref* are describing the intra-level instantiation on the level of class syntax, class semantics, object syntax and object semantics.

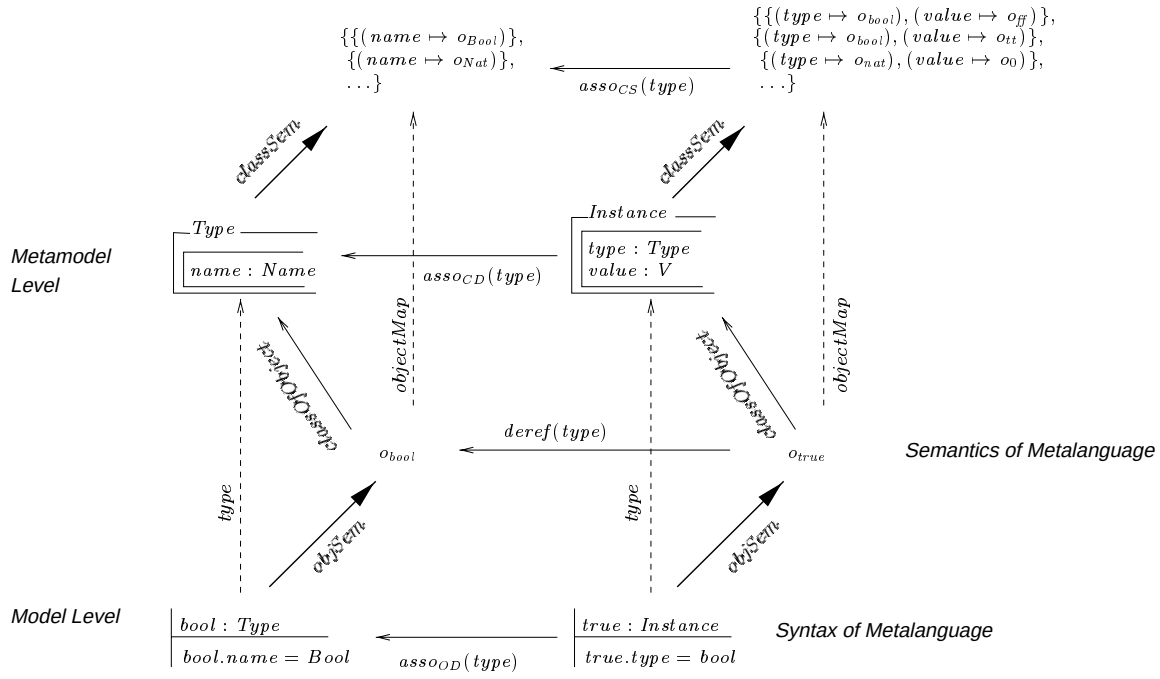



Figure 8: Example

In Figure 8, an example of an intensional (*Type*) and a related extensional entity (*Instance*) on the metamodel level is depicted. These are instantiated on the model level to the objects *bool* and *true*. Accordingly, *true* is an extension of *bool* on the model level. The properties apply, of course, to all pairs of intensional and extensional objects. Again, the proofs can be found in [GKP98].

- The semantics of an object is compatible with the semantics of the object's class.
- Every association between classes is reflected by a compatible relation between valuations on the semantical level.
- The instantiation relation between an extensional and an intensional object on the metamodel level is reflected by a compatible relation between corresponding extensional and intensional objects on the model level, i.e. the intra-level instantiation is also instantiated by an inter-level instantiation.
- Associations between semantics objects on the model level are compatible with the class semantics of the participating objects.

Note that most other metamodeling approaches (e.g. [UML97, MOF97, Atk97]) do not consider inter-level instantiation. Thus the relation between metamodel level and model level is not precisely defined in these approaches.

# 5 Discussion

In the previous sections we have discussed our metamodeling technique. The major contributions of this work are the following: We advocate the usage of a formal metalanguage and we elaborate the concepts the metalanguage should provide. Based on these facilities of the metalanguage, we show how the these techniques can be successfully applied in the definition of metamodels. Here, one of the major aspects is the distinction between intensional and extensional entities. Both kinds of entities and their instantiations have been investigated in detail in Section 4. In contrast to existing metamodels, we also consider the dynamic semantics of a metamodel.

## 5.1 Benefits of the presented approach

One benefit of our approach is the suitability for the precise definition of multiple view languages. In addition, our approach can be useful for several other areas in software engineering. Using the proposed technique we are able to define a complete model of the system that comprises not only the intensional parts but also the extensional ones. Transformation of models is naturally expressed by intensional-object-manipulating methods. I. e. adding a transition to an automaton would be described by a method *AddTransition* which manipulates the attribute *transitions*. Furthermore, context conditions for model transformation can be expressed, e. g. the impact of a change request in one model on other models can be specified. As we have shown in Section 3, this enables us to define system evolution as well as the evolution of objects. The dependencies, the impact of changes and the system evolution invariants can be expressed. Semantic preserving model transformations are also in the scope of the presented approach. Using our technique, we are able to define the conditions under which a model transformation is semantically correct.

Methodological rules or style guides can be incorporated into the metamodel. This means that even if a model is syntactically and semantically correct, it might not be conform to some further conventions (e. g. naming conventions), or it might include some unwanted nondeterminism, some methods might be partially defined and cannot be called in every state of an object etc. This important information can be expressed using our approach.

Tracing through different development phases can be expressed by storing the history of a model element. In this way, the evolution of a modeling object might be traced from analysis down to implementation.

### 5.2 Comparison to other Approaches

The *Unified Modeling Language* is a set of object-oriented modeling languages [UML97]. The relationship between the modeling languages is described by a metamodel [USE97]. Technically, the abstract syntax of UML is given by means of class diagrams, the consistency constraints are defined by OCL [OCL97] formulas, and the dynamic semantics is explained in natural language. We believe that the UML is very important progress towards a precise understanding of object-oriented modeling techniques. In particular, our work [MK98] is based on it. In contrast to the UML, we use a single formal metalanguage for the definition of object-oriented modeling languages. In this way, we avoid self-referencing problems and achieve a precise semantics.

The *Meta Object Facility* (MOF) [MOF97] is a meta-metamodel (a model for the definition of metamodels). Its key goal is to provide means for extensionality and self-discovery in systems. It addresses the manipulation of meta-data, e.g. the creation and retrieval of metaentities. MOF is described by a combination of UML notations, CORBA interfaces, explanatory text and constraints given in OCL. Our approach is also applicable for the definition of a meta-metalanguage, like MOF. Basically, the advantage of our approach is the utilization of a formal language for the definition of two layers, as well as the precise definition of dynamic semantics.

*Atkinson* [Atk97] describes a metamodeling framework. He also identifies the problem of an entity being viewed as both instance and template[4] and therefore introduces the concept of *clabject*. A clabject is comprised of an intensional and its extensional entity. The instantiation of clabject is not covered. As an additional difference to our work, Atkinson claims that operations and methods are unimportant for metamodeling in practice, while we advocate their benefit for the description of system dynamics and evolution. The strictness of layers is reflected in our approach by the strict separation of the intensional and extensional objects in every level of abstraction.

In [PBF98], a conceptual model representing the information acquired during object-oriented analysis and design is presented. This conceptual model integrates entities and metaentities into a single conceptual framework based on Dynamic Logic. The authors propose a transformation method consisting of a set of rules for systematically creating a single integrated dynamic logic model from the separate elements that constitute a description of an object-oriented system expressed in the Unified Modeling Language. The intended semantics for this conceptual model is a set of states with a set of transition relations on states. The domain for states is an algebra whose elements are both entities and metaentities. The set of transition relations is partitioned into two disjoint sets: a set of transition representing modifications on the specification of the system (i. e. evolution of metaentities), and a set of transition representing modifications on the system at run time (i. e. evolution of entities).

The *PUML group* [EFLR98] aims at a precise semantic model for UML class diagrams. This semantics model shall be used as a basis for the definition of (semantic preserving) transformation rules. It is argued that a metamodel cannot serve as a precise description of the meaning of UML constructs, but rather as a precise desription of the UML notation. The

[4] *Template* and *Instance* correspond to intensional and extensional objects in our work.

authors describe abstract syntax, semantic domain and a mapping from syntax to semantics in the specification language Z. The similarities between their approach and the one presented in this paper are thus the identification of a dichotomy and a single-language approach. Inter-level instantiation could be also expressed in the PUML approach, though it is not explicitly formulated. The major difference between the two approaches is that no dynamic semantics is described in [EFLR98], and that we apply a meta language that supports a reference semantics, while in the PUML approach, the meta language supports value semantics only.

### 5.3 Metamodeling and Denotational Semantics

We believe that there is a strong relation between metamodeling and denotational semantics. In both disciplines, the concepts of abstract syntax, static and dynamic semantics play a central role.[5]

In the denotational semantics area, the abstract syntax is defined by abstract grammars, the static semantics is defined by validity functions, checking the well-formedness of the syntactic constructs of the abstract grammar. The dynamic semantics (denotation) is defined by assigning mathematical objects from the semantics domain (usually sets and functions) to the syntactic constructs. It is remarkable that the signature of the semantic functions usually comprises an explicit state and delivers a state. This is necessary because, in set theory, there is no explicit notion of state.

| | Abstract Syntax | Static Semantics | Dynamic Semantics |
|---|---|---|---|
| Metamodeling | Class diagrams Object-Z classes & attributes | Object-Z state invariants & secondary attributes & axiomatic definitions | Object-Z methods |
| Denotational Semantics | Abstract Grammars | Validity conditions for constructs of the abstract grammar | Meaning Functions from constructs of abstract grammar into semantic domain |

Figure 9: Metamodeling and Denotational Semantics

In the presented approach, the abstract syntax of a language is reflected by the attributes and associations of the extensional objects. The static semantics is given by additional secondary attributes and constraints on the attributes of the class. The dynamic semantics is given by methods manipulating the extensional objects. The extensional objects are the correspondents to the state in the denotation functions. A comparison between our approach to metamodeling and denotational semantics is given in Figure 9.

We believe that these parallels have not become apparent so far because of the neglect of dynamic semantics in metamodeling. In our opinion, the presented approach offers essential advantages to the denotational semantics approach because it provides the full power of object-orientation, instead of simple set theory, for the description of the semantic domain.

[5] We would like to point out that the presented approach is not a transformational one, i. e. we do not aim at translating models in different languages into one single language (the metalanguage), but we identify the concepts common to the different languages and express them in the metalanguage.

# References


[Atk97] Colin Atkinson. Meta-modeling for distributed object environments. In *First International Enterprise Distributed Object Computing Workshop, EDOC'97*. IEEE Computer Society Press, 1997.

[CDI+97] Stephen Crawley, Scott Davis, Jaga Indulska, Simon McBride, and Kerry Raymond. Meta-meta is better-better! In *Workshop on Distributed Applications and Interoperable Systems (DAIS), Cottbus, Germany*, September 1997.

[CN93] M. R. Cohen and E. Nagel. *An Introduction to Logic*. Hackett Publishing Company, Indianapolis, Indiana, 1993.

[Don96] Jin Song Dong. *Formal Object Modelling Techniques and Denotational Semantics Studies*. PhD thesis, Department of Computer Science, University of Queensland, May 1996.

[DRS94] Roger Duke, Gordon Rose, and Graeme Smith. Object-z: a specification language advocated for the description of standards. Technical Report 94-45, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, December 1994.

[EFLR98] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Developing the uml as a formal modelling notation. In Pierre-Alain Muller and Jean Bezivin, editors, *UML'98 Beyond the notation. International Workshop Mulhouse France*. Ecole Superieure Mulhouse, Universite de Haute-Alsace, 1998.

[Gei98] Robert Geisler. *Formal Semantics for the Integration of Statecharts and Z in a Metamodel-Based Framework*. PhD thesis, Technical Universiy of Berlin, Oct 1998. submitted.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[GKP98] R. Geisler, M. Klar, and C. Pons. Dimensions and dichotomy in metamodeling. Technical Report 98-5, TU Berlin, Fachbereich Informatik, 1998. http://tfs.cs.tu-berlin.de/~espress/doc/own/GKP98.ps.

[Mey88] B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, 1988.

[MK98] S. Mann and M. Klar. A metamodel for object-oriented statecharts. In *The Second Workshop on Rigorous Object-Oriented Methods, ROOM 2, Bradford*, May 1998.

[MOF97] *Meta Object Facility (MOF) Specification*, October 1997. Joint revised submission to the Object Management Group (OMG), http:/www.omg.org.

[OCL97] *The Object Constraint Language – Version 1.1*, September 1997. Part of [UML97].

[Ode95] James Odell. Meta-modeling. In *OOPSLA'95 Workshop on Metamodeling in OO*, October 1995.

[Öve98] Gunnar Övergaard. A formal approach to relationships in the unified modeling language. In *PSMT – Workshop on Precise Semantics for Software Modeling Techniques in conjunction with ICSE 98, Kyoto, Japan*, 1998.

[PBF98] C. Pons, G. Baum, and M. Felder. Integrating object-oriented model with object-oriented metamodel into a single formalism. In *Second ECOOP Workshop on Precise Behavioral Semantics, Brussels*, July 1998.

[UML97] *The Unified Modeling Language (UML) Specification – Version 1.1*, September 1997. Joint submission to the Object Management Group (OMG), http://www.rational.com/uml/.

[USE97] *UML Semantics – Version 1.1*, September 1997. Part of [UML97].