UNIVERSITÄT KOBLENZ · LANDAU

A Booch Metamodel

Roger Süttenbach, Jürgen Ebert

5/97

Universität Koblenz-Landau
Fachbereich Informatik

Fachberichte
INFORMATIK

Universität Koblenz-Landau
Institut für Informatik, Rheinau 1, D-56075 Koblenz
E-mail: researchreports@infko.uni-koblenz.de,
WWW: http://www.uni-koblenz.de/fb4/

A Booch Metamodel

Roger Süttenbach, Jürgen Ebert
University of Koblenz-Landau
Institute for Software Technology
{ sbach, ebert }@informatik.uni-koblenz.de

Abstract

Object-oriented methods, like the Booch method, are widely used in the development of software systems nowadays, but their syntax and semantics are only defined by natural language text and examples. This paper provides a formalized description of the syntax of the Booch method by using the EER/GRAL approach of modeling.

Keywords: EER/GRAL, object-oriented methods, declarative modeling, Booch method

1 Introduction

1.1 Intention of This Paper

Object-oriented analysis and design methods, often called object-oriented methods for short, are more and more frequently used in the development of software systems nowadays. One of the best-known methods is the *Booch method* described in [B94]. The essential part of this book is the description of a visual language used for modeling real systems. The language defines what diagrams of real systems look like and whether they are correct in the sense of the language or not. Additionally, the book explains the use of the language in the development process and it provides a lot of example diagrams applying the language to real systems. There are two main problems concerning the description of the Booch language for modeling:

- its lack of formalization, and
- its lack of integration.

The *lack of formalization* refers to the unformalized basis of the language. Its concrete elements are icons which can be connected to some other icons. Their meanings and the possible relations between them are described by natural language text and examples. Therefore, decisions whether diagrams which model a real system are correct or not, can only be made on account of this informal description. Since most diagrams of real systems differ a little from those examples, these decisions are often ambiguous and vague. Beyond this, the correctness of a diagram depends on certain integrity conditions. These integrity conditions are not part of the icons and are only described by text.

The *lack of integration* denotes the fact that a visual language often consists of several unconnected parts referring to different views of a system. The relations between these parts are mostly not described but only illustrated by rules of thumb. Even the relations within these parts are not shown in a complete presentation. A complete and coherent description of the entire language as a whole does not exist. It is not possible to look at a certain icon and directly to identify its possible relations with other icons. The quick reference in the book [B94], which is intended to achieve this, only provides a reduced view of the language. It does not show the possible relations between the individual icons and it does not show any constraints concerning the correctness of diagrams.

In order to deal with these two important problems it is necessary to produce a formalized description of the visual language, or in other words to make a *metamodel* of it. Hereby, we use the EER/GRAL approach of modeling which we explain in section 1.3 in more detail. An EER/GRAL description defines the set of *syntax graphs* of the diagrams used. That description is called a metamodel. This term refers to the fact that instances of this model are models themself. A syntax graph of a Booch model does not describe the concrete syntax of this model but for every model, expressed by Booch diagrams, there is only one abstract syntax graph. This syntax graph can be checked whether it is a correct graph of the Booch metamodel or not. Thus, a Booch diagram is a correct diagram if its corresponding syntax graph is an instance of the Booch metamodel.

1.2 Modeling Principles

In this section we want to highlight some principles which form the basis of constructing a metamodel of a visual language. These principles can be seen as the basic guidelines of our modeling. If there are several possibilities of modeling, they help us to select the best model in the sense of these principles.

Principle of Correctness
The metamodel must be a correct model of the language. This refers to two kinds of correctness. First, the metamodel should prohibit the possibility of generating incorrect diagrams. Second, the metamodel should permit the possibility of generating all correct diagrams.

Principle of Completeness
The essence of a visual language should be completely expressed in the corresponding metamodel. This includes the concepts, the relations between them, and possible constraints. Constraints may describe global restrictions which must be preserved at any time, or local restrictions which result from certain constellations.

Principle of Clarity
A metamodel must reproduce or even produce a structured view to a language. I.e. similar concepts of the language must be represented in a similar way in the metamodel. Clarity also means to make information explicit. Hidden dependendencies or restrictions must be modelled in a straight and clear way, and should not be left out.

Principle of Simplicity
Setting up a metamodel of a visual language means providing a pure view without any redundant information. Not every concept, which is mentioned in a language, is really a new or independent concept. Information is often the same in different contexts, but it shall be modelled only once.

There are some *contradictions* and some *overlappings* among these principles. For example, the principle of completeness competes with the principle of simplicity concerning redundancy, as do the principles of simplicity and correctness. An example of overlapping is the explicit modeling of constraints which simultaneously supports the principles of completeness, clarity and correctness. The same applies to the principles of clarity and simplicity because clear things are often simple things, and vice versa. In case of doubt, we refer to the following *order of principles* (enumerated from the most important to the least important):

1. principle of correctness,
2. principle of completeness,
3. principle of clarity, and
4. principle of simplicity.

1.3 Modeling Building Blocks

In order to formalize the abstract syntax of a method one needs a formal basis. This paper uses the *EER/GRAL approach* of modeling using TGraphs as described in [EWD+96] and [EF94]. Here, Graphs are used to achieve a formal modeling which has been applied to various fields in software engineering. Graph classes – which are sets of graphs – can be defined with *extended entity relationship (EER) descriptions* which are annotated by integrity conditions expressed in the *constraint language GRAL* (GRAph specification

Language). The ERR/GRAL description in this paper defines the set of correct syntax graphs.

Before we go on with the description of the Booch metamodel, we would like to sketch the two parts of the description approach itself – namely EER diagrams and GRAL predicates.

1.3.1 EER

EER descriptions [CEW95] may contain five different modeling building blocks – *entity types*, *relationship types*, *attributes*, *generalizations*, and *aggregations*. Regarding the underlying graph approach an entity type defines a set of vertices whereas a relationship type defines a set of edges for an instance graph. An attribute adds additional information on vertices or edges. Generalization defines a hierarchy between vertex types whereas aggregation can be chosen to add structural information.

EER descriptions have the *visual representation* as shown in figure 1:

Entities[1] are represented by hollow rectangles in which the name of the entity is positioned.

Relationships are represented by lines with the name attached to it. The arrow on the line determines the read direction of the name with regard to the connected entities. The arrowheads at the end of the line denote cardinality.

Attributes are represented by round boxes connected to entities or relationships, or they are annotated within the entities. Attributes are typed. They may have basic types such as STRING and INTEGER, or enumeration types recognizable by the curved parantheses.

Generalization is expressed by a *Venn notation* which means that the specialized entities are drawn within the general entities. A filled rectangle marks an abstract entity type which has no instances.

Aggregate entities are adorned by a diamond which connects the aggregate to its components. It may be possible to assign an entity to several aggregate entities.

EER descriptions *are used to formalize* the language in the following way:

Entities are used to describe concepts of the modelled method. That means they represent information which cannot be derived or built from other concepts. Entities are mainly used to meet the principle of completeness.

Relationships express which concepts can be connected to each other. They support the principle of simplicity because the quantity of the items is reduced and they support the principle of clarity because the connections are denoted explicitly.

Attributes are additional pieces of information. The properties of concepts and connections are described by attributes. They describe certain

[1] Further on, we take the term *entities* instead of *entity types* and *relationships* instead of *relationship types* because it is easier to read.

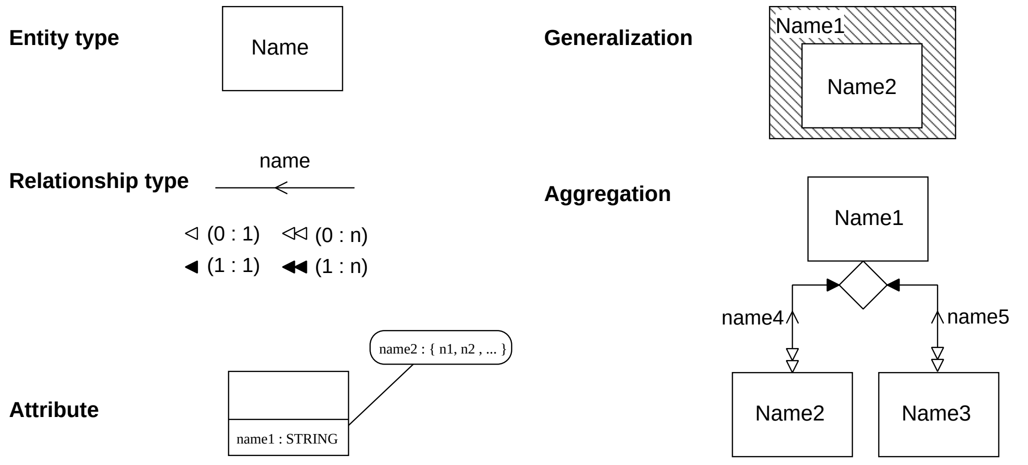


Figure 1: Modeling Building Blocks

aspects and are not concepts themselves. Note that in our notation relationships may have attributes too. Attributes strongly support the principle of simplicity because they make information local.

Generalization is used to refine existent concepts, or to combine them into new ones if the concepts are similar with respect to attributes or relationships. Generalization supports the modeling principles of simplicity and of clarity because it reduces redundant information and emphasizes similarity.

Aggregation allows ternary and higher relationships to be factored into binary ones. This makes the model simpler to understand because of the separation of concerns. Aggregation is used to support the modeling principle of clarity because of emphasizing certain structures of the concepts.

1.3.2 GRAL

In the EER/GRAL approach the $\mathcal{Z}$-like [S92] *assertion language GRAL* is used to denote such integrity conditions which cannot be expressed by the EER description. These may be constraints on the values of the attributes of vertices and edges, the existence or non-existence of a certain path in a graph, the cardinality restriction of vertex sets depending on attribute values, etc.

A GRAL assertion is a sequence of predicates which directly refer to the corresponding EER description. The syntax and the semantics of GRAL is described formally in [F96]. GRAL predicates can be tested efficiently.

Here we do not describe GRAL in detail but we give some examples in the context of the Booch method and some hints on how to use them.

Restrictions on certain vertices, edges or attributes can be expressed easily as the following example[2] shows:

Constraint Each subsystem name in a system has to be unique.

$$\mathsf{MD2}: \quad \forall\, sb_1, sb_2 : V_{Subsystem} \mid sb_1 \neq sb_2 \bullet sb_1.name \neq sb_2.name \;;$$

This means each vertex of type Subsystem must have a different value in its attribute name, or in other words the values in the attribute name must differ if they belong to different vertices.

❋

Restrictions on sets of vertices depending on the existence of paths in graphs can be described in the following way:

Constraint Categories are only allowed to have using relations.

$$\mathsf{CD7}: \quad \forall\, cg : V_{Category} \bullet (cg \rightharpoonup_{isFirstIn} \cup\, cg \rightharpoonup_{isSecondIn}) \subseteq V_{UsingRelation} \;;$$

This means for all vertices of type Category the set of vertices which is reachable via a direct outgoing edge from type isFirstIn or from type isSecondIn must be a subset of the set of all vertices of type UsingRelation.

❋

Constraints on the existence or non-existence of paths in graphs which imply reachability restrictions can be expressed as follows:

Constraint Cycles are not allowed in the inheritance hierarchy.

$$\mathsf{CD9}: \quad \forall\, v : V_{ClassUnit} \bullet \neg\, (c\ (\rightharpoonup_{isFirstIn} \bullet_{Inheritance} \leftharpoonup_{isSecondIn})^{+}\, c) \;;$$

The GRAL predicate above requires that no vertex of type Class Unit is the starting point of a path via outgoing edges of type isFirstIn, vertices of type Inheritance and incoming edges of type isSecondIn back to itself. The symbol '+' denotes that at least one but may be several loops are regarded.

The predicate is an example for *regular path expressions* in GRAL which are one of the most powerful features in describing the constraints to graph classes. They are regular, i.e. they are built as sequences, iterations, or alternatives of other path expressions, which can be regular or atomic ones.

❋

In GRAL it is possible to combine several matching constraints in a single GRAL predicate. Furthermore, GRAL provides a library with predefined functions. For example, *degree* returns the number of incoming or outgoing

[2] The abbreviation in front of the GRAL predicate refers to the corresponding place in this paper. For example, MD2 refers to the second constraint in the module diagram.

edges regarding a certain vertex:

Constraint If a superstate has a direct transition going to it, one of the direct nested states must be a start state.

$$\mathsf{STD6}: \quad \forall\, su : V_{Superstate} \mid degree(\leftharpoonup_{goesTo}, su) > 0 \\ \bullet\, (\exists_1\, s : V_{State} \bullet su \rightharpoonup_{contains} s \wedge s.flag = start) \;;$$

Here, the GRAL predicate requires that for all vertices of type Superstate with incoming edges of type goesTo there must be another vertex of type State which has the value 'start' in its attribute flag and is connected to its superstate by an outgoing edge of type contains.

Structure of This Paper

In chapter **2 Logical View**, **3 Dynamic View** and **4 Physical View** we incrementally describe each part of the Booch method. We are guided in this order by the quick reference presented on the first and last two pages of the book [B94]. Each icon in this quick reference has an EER description as its counterpart, possibly followed by several GRAL predicates below. Beyond this, we give some explanations of the icons' meaning and their graphical representation above the figure, and explanations with respect to the EER description below it. In order to distinguish previously mentioned concepts from newer ones we draw those in a lighter gray. At the end of each chapter the individual EER/GRAL descriptions are merged into a single description for the whole view.

There are two problems with this procedure. First, the quick reference is not always complete because many elements of the language are left out there. For example, in state transition diagrams the entry- and the exit-action are not mentioned in the quick reference but in the book on page 204. We deal with this problem in such a way that the quick reference is extended to the complete notation. Second, the assignment of the diagrams to the logical, dynamic or physical view is not always obvious. Class diagrams surely belong to the logical model but for example, object diagrams can be assigned to the logical and the dynamic view as well. We choose an assignment which reflects the basic idea of the diagram. But that is not really a problem because it is only a logical assignment which has no influence on the metamodel as a whole.

In the **Appendix** we summarize the three views and a collection of the constraints to an **Overall Metamodel** for the method as a whole.

2 Logical View

In the Booch method the *logical view* of a system is described by the class structure of this system. The class structure is represented by two sets of documents:

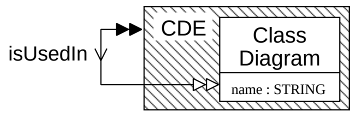
- a set of class diagrams, and
- a set of specifications.

A *class diagram* represents the class structure in a mainly graphical way whereas a *specification* represents the class structure by natural language text only.

2.1 Class Diagram

A *class diagram* is a named sheet of paper showing one or more icons which represent all or a part of the class structure of a system.

A class diagram has no explicit icon in the Booch method. One may take the sheet of paper as the graphical representation.



A Class Diagram comprises all elements which are assigned to it by the isUsedIn relationship. Here, the abstract concept class diagram element, CDE for short, is used as a placeholder for these elements which have to be specialized concepts of the CDE. For graphical simplicity, this is done in figure 2 (p. 22) at first.

Constraint The name space of class diagrams has to be unique in a system.

CD1 : $\forall\, cd_1, cd_2 : V_{ClassDiagram} \mid cd_1 \neq cd_2 \bullet cd_1.name \neq cd_2.name$;

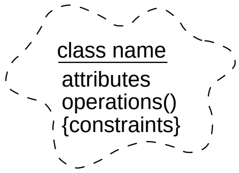
❋

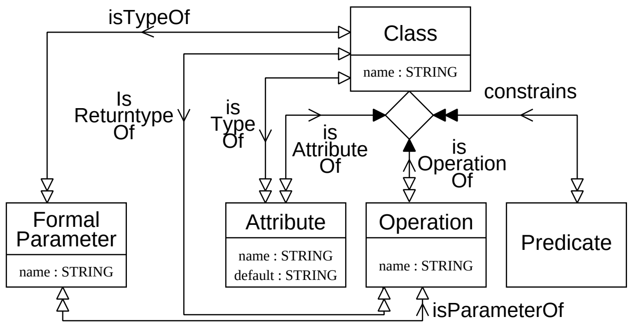
Class Icons

A *class* defines the properties and the behaviour of a set of similar objects; an "object has state, behaviour and identity" (p. 83)[3].

A class is represented by a cloud icon. Within this cloud the name, the attributes, the operation signatures and the constraints of this class are listed one after the other. In order to distinguish them the following conventions are given: the name is separated by a line from the other ones, the operations are marked by round parentheses containing their formal parameters, and the constraints are marked by curved parentheses.

[3] The page numbers refer to the book [B94].



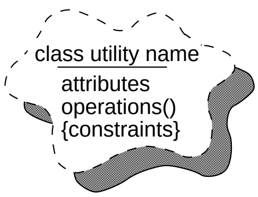


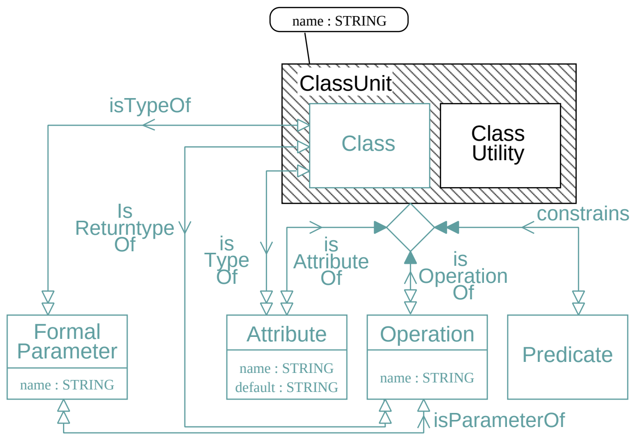
The modeling of the concept Class refers to the description above. Properties and behavior are expressed by Attributes and Operations. An attribute may be additionally described by its type and it may have a default value. The description of a signature of an operation is guided by the C++ syntax. Each operation possibly has a return class and has zero or more Formal Parameters having a class associated as their type. A constraint is defined by Booch as "an expression of semantic condition that must be preserved" (p. 193) and he uses Predicates as constraints. The concept is not further refined here. In general, constraints look like "restart time $>=$ 5 minutes" (p. 193) or "delivery $<=$ 48 hours" (p. 405).

❄

A *class utility* is a logical construct with two possible meanings: first, it is a collection of operations free of an affiliation to a class or second, it is a class that only has class attributes and operations. Class attributes are such attributes which are not assigned to the objects of a class but to the class itself.

A class utility is represented by a cloud icon with an additional shadow. The possible items within the cloud icon for class utilities are identical with the entries within the cloud icon for classes.





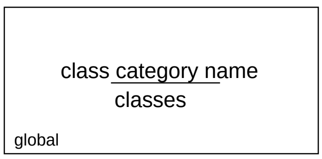
Classes and Class Utilities are almost identical with respect to their relationships, so the generalization is used to model this aspect. The abstract concept

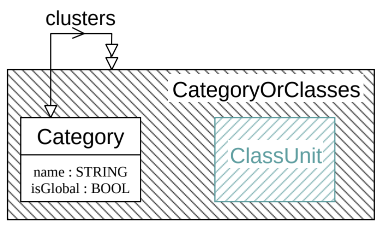
Class Unit is an artificial concept which is introduced to make the metamodel simpler. Therefore, the property name of the concept Class is shifted to the concept Class Unit.

❋

Class categories are a way of logically clustering the model by assigning classes to categories. This can be done under different aspects of interest, for example in order to cluster all classes which are needed to fullfil a certain task. Further tasks of a category are the control of name space and the control of visibility. A class category "represents an encapsulated name space" (p. 182), i.e. a name has to be unique with respect to its category but not with respect to the whole system. The control of visibility is accomplished because a category must be imported[4] by a class, if its classes are used in this class. If a category is global, then the import of its classes is done by default.

A class category is represented by a rectangle. The class category name is annotated within the rectangle. The classes which belong to the category are listed below a line within the rectangle. A global category is annotated by the keyword 'global'.





Class categories are modelled by the concept Category. The information, which class belongs to a certain class category, is expressed by the relationship clusters. Note that a category may cluster other categories as well and that a class does not have to be clustered by a category but can only be clustered in one.

There are several constraints connected with the use of categories.

Constraint Each class category name has to be unique in a system.

$$\mathsf{CD2}: \quad \forall\, cg_1, cg_2 : V_{Category} \mid cg_1 \neq cg_2 \;\bullet\; cg_1.name \neq cg_2.name\ ;$$

Constraint Each class category name in a system has to be distinct from all other class names.

$$\mathsf{CD3}: \quad \forall\, cg : V_{Category};\ c : V_{ClassUnit} \;\bullet\; cg.name \neq c.name\ ;$$

[4] Import of classes is described in section **Class Relationships** on page 12.

Constraint The class name space in a system – with respect to each category and with respect to the toplevel – has to be unique[5].

$$\text{CD4}: \quad \forall\, c_1, c_2 : V_{ClassUnit} \mid c_1 \neq c_2 \bullet c_1.name \neq c_2.name \vee (\exists\, cg : V_{Category} \bullet cg \rightharpoonup_{clusters} c_1 \wedge \neg\, (cg \rightharpoonup_{clusters} c_2))\ ;$$

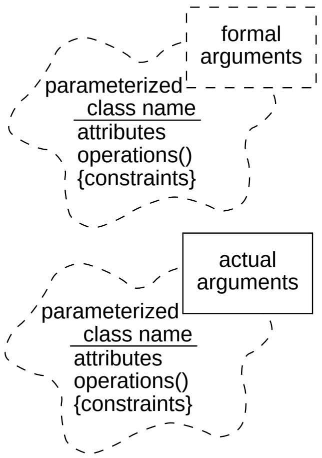
Constraint The classes and categories assigned to a category must be ordered in a hierarchy.

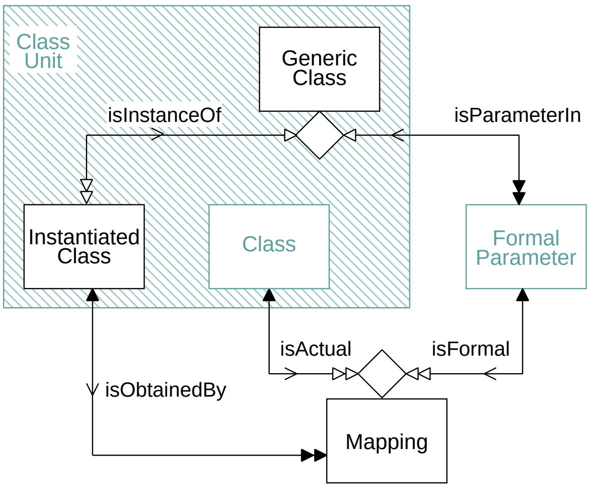
$$\text{CD5}: \quad isForest(eGraph(\rightharpoonup_{clusters}))\ ;$$

❋

Parameterized classes are templates for other classes which are generic with respect to certain properties using formal parameters. The instances of such a template are obtained by providing actual parameters for each of the formal parameters.

A parameterized class is represented by a cloud icon with an additional rectangle and the formal parameters which are listed within the rectangle. The instances are represented in the same way but with a solid rectangle and the actual parameters within it.





Parameterized classes are described by the more commonly used concept Generic Class which has one or more Formal Parameters and has as instances zero or more Instatianted Classes. The instantiated class is Obtained By a Mapping from each of the formal parameters, expressed by the isFormal relationship, to actual parameters, respectively classes, expressed by the isActual

[5] In other words, if two different classes have the same name, at least one must belong to a different category.

relationship.

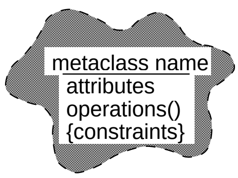
Constraint For each instantiated class there must be a complete mapping from each formal parameter to an actual parameter with respect to its generic class.

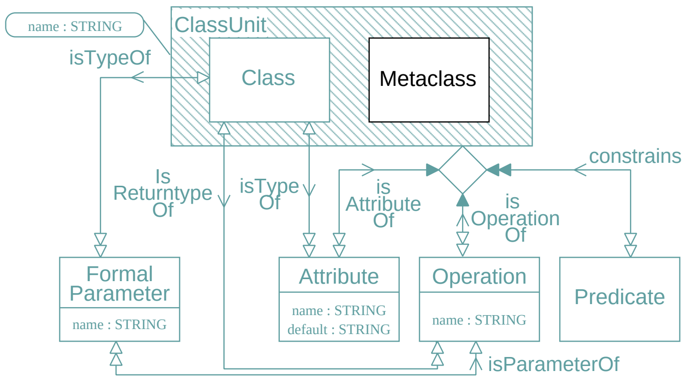
$$\text{CD6}: \quad \forall\, ic : V_{InstantiatedClass};\ gc : V_{GenericClass} \mid ic \rightharpoonup_{isInstanceOf} gc$$
$$\bullet\ (\forall\, fp : V_{FormalParameter} \mid fp \rightharpoonup_{isParameterIn} gc$$
$$\bullet\ fp \rightharpoonup_{isFormal} \leftharpoonup_{isObtainedBy} ic\ )\ ;$$

❋

A *metaclass* is the class of a class. It is used to express information which refers to the objects of that class in their entirety. For example, constructor operations or class attributes can be described in metaclasses.

A metaclass is represented by a filled cloud icon. The possible entries are similar to the entries for classes.





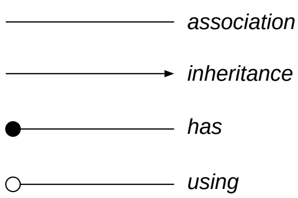
A metaclass "may not itself have any instances, but may inherit from, contain instances of, use, and otherwise associate with other classes" (p. 185). Therefore, we model the Metaclass as an refined concept of Class Unit by using generalization.

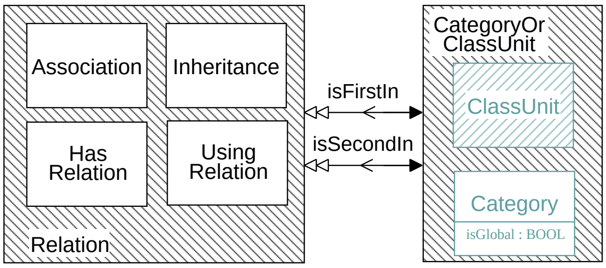
❋

Class Relationships

An *association* "denotes a semantic connection" (p. 179) between two classes. These semantics are not further refined whereas an *inheritance*, a *has* and a *using relation* have special semantics. The inheritance and the has relations express hierarchies. The inheritance relation describes the generalization hierarchy and the has relation describes the aggregation hierarchy. The using relation describes that one class uses certain services – attribute values or operations – from other classes.

All relations are represented by a single line possibly with additional adornments at its start or at its end. An association has no adornment whereas an inheritance relation has an arrow pointing to the generalized class. The has relation is adorned with a filled circle near the aggregate class and the use relation has a hollow circle near the class which uses the services of the other class.





The four relations are modelled by the corresponding concepts Association, Inheritance, Has Relation and Using Relation. The abstract concept Relation is used to simplify the model. Each relation has exactly one start and one end class which is expressed by the isFirstIn and the isSecondIn relationships. By convention the start class is the specialized class in an inheritance relation, the aggregate class in a has relation and a using class in the using relation. The start class in an association has no special meanings, it can be one of the two classes.

The using relation may also apply to Categories but then it has the semantics of *import* of the classes of this category which makes the classes visible to other classes.

Constraint Categories are only allowed to have using relations.

CD7 : $$\forall cg : V_{Category} \bullet (cg \rightharpoonup_{isFirstIn} \cup cg \rightharpoonup_{isSecondIn}) \subseteq V_{UsingRelation} \ ;$$

Constraint If a class uses, inherits from, contains instances of, or associates with another class then:
both classes are on top of the system,
or the category of the second class is a global category,
or both classes are in the same respectively deeper clustered category,
or there is a using relation between the categories which cluster the two classes.

CD8 : $$\begin{aligned} &\forall c_1, c_2 : V_{ClassUnit} \mid c_1 \rightharpoonup_{isFirstIn} \leftharpoonup_{isSecondIn} c_2 \\ &\quad \bullet (degree(\leftharpoonup_{clusters}, c_1) = 0 \wedge degree(\leftharpoonup_{clusters}, c_2) = 0) \vee \\ &\quad (\exists cg : V_{Category} \bullet cg(\rightharpoonup_{clusters})^+ c_2 \wedge cg.isGlobal = TRUE) \vee \\ &\quad (\exists cg : V_{Category} \bullet cg(\rightharpoonup_{clusters})^+ c_1 \leftharpoonup_{clusters} (\rightharpoonup_{clusters})^+ c_2) \vee \\ &\quad (c_1(\leftharpoonup_{clusters})^+ \rightharpoonup_{isFirstIn} \bullet_{UsingRelation} \leftharpoonup_{isSecondIn} (\rightharpoonup_{clusters})^+ c_2) \ ; \end{aligned}$$

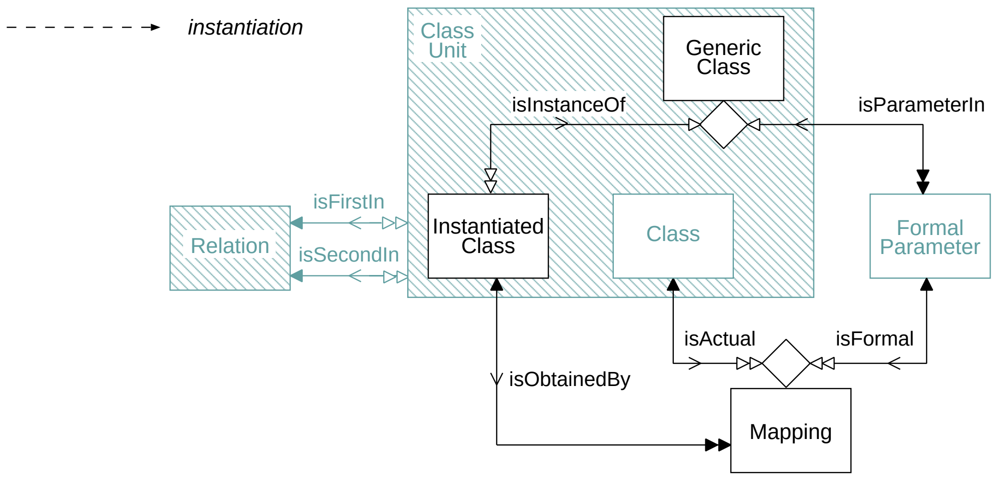
Constraint Circles are not allowed in the inheritance hierarchy.

$$\mathsf{CD9:} \quad \forall\, v : V_{ClassUnit} \bullet \neg\, (c\ (\rightharpoonup_{isFirstIn} \bullet_{Inheritance} \leftharpoonup_{isSecondIn})^{+} c)\ ;$$

❋

An *instantiation relation* describes that a certain class is an instance of a parameterized class[6].

An instantiation relation is represented by a dashed line with an arrow at the end pointing to the parameterized class.



The instantiation is already modelled in the aggregation Generic Class by the isInstanceOf relationship. But there is an additional constraint which must be preserved.

Constraint There must be a using relation or an association between the instantiated class and the classes used as actual parameters.

$$\begin{aligned}\mathsf{CD10:} \quad & \forall\, ic : V_{InstantiatedClass} \\ & \quad \bullet (\forall\, m : V_{Mapping} \mid ic \rightharpoonup_{isObtainedBy} m \\ & \qquad \bullet (m \leftharpoonup_{isActual} (\rightharpoonup \bullet_{Association} \leftharpoonup \mid \\ & \qquad \rightharpoonup_{isSecondIn} \bullet_{UsingRelation} \leftharpoonup_{isFirstIn})\ ic))\ ;\end{aligned}$$

❋

A *metaclass relation* describes that a certain class is the metaclass of this class.

A metaclass relation is represented by a gray line with an arrow pointing to the metaclass.

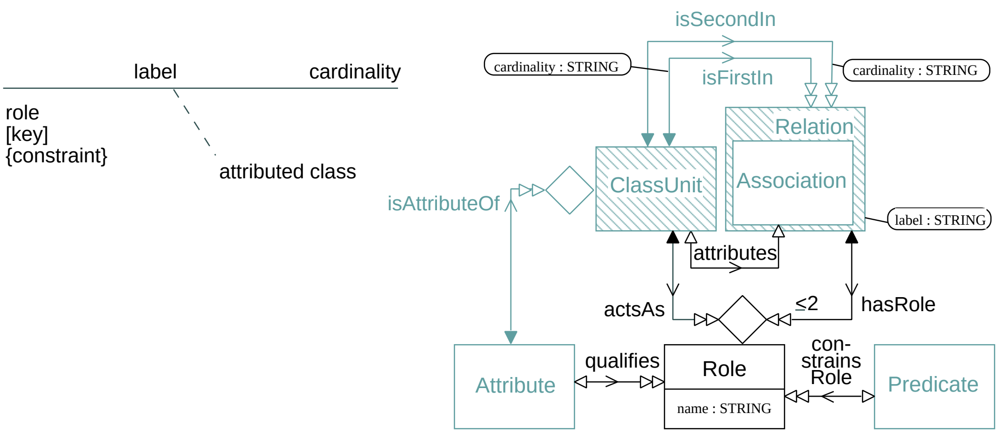
[6] This is needed because in the Booch method the corresponding parameterized class cannot be referred by its name.

The metaclass relation between a Class and its Metaclass is modelled by the isMetaclassOf relationship between them.

Relationships Adornments

The relations between two classes can be refined by various *textual information*. First of all, the relation can be labeled by a name. Then, the role of each class in the relation can be annotated too. A role describes the special context in which the corresponding class acts in this relation. The cardinality expresses the number of possible links modelled by the relation with respect to each class. A key is defined as "an attribute whose value uniquely identifies" (p. 193) an object. The key is used to navigate in a set of objects denoted by an association. A constraint is a predicate about the relation referring to one or to both of the classes. An attributed association is an association which has a class as an attribute.

The information is represented as natural language text. Additionally, the key is enclosed by square parentheses and the constraint by curved parentheses. Information which refers to a certain class in the relation are positioned near this class. The attributed class is connected to the relation with a dashed line.



The name of the relation is modelled by its label. The Role always refers to one Class Unit and one Relation in which the class acts As a specific role described by the role name. A key is an Attribute which qualifies the association indirecly by its Role. Similarly a constraint is modelled by a Predicate which constrains the Role . The attributed association is modelled by a ternary association in which a certain class attributes the Association.

Constraint The key must be an attribute of the other class connected with the relation.

$$\text{CD11}: \quad \forall\, ro : V_{Role};\ a : V_{Attribute};\ c : V_{ClassUnit} \mid a \rightharpoonup_{qualifies} ro \leftharpoonup_{actsAs} c \\ \bullet\, (\exists\, c_1 : V_{ClassUnit};\ r : V_{Relation} \mid r \rightharpoonup_{hasRole} ro \\ \bullet\, (c \rightharpoonup_{isFirstIn} r \leftharpoonup_{isSecondIn} c_1 \leftharpoonup_{isAttributeOf} a) \vee \\ (c \rightharpoonup_{isSecondIn} r \leftharpoonup_{isFirstIn} c_1 \leftharpoonup_{isAttributeOf} a)) \ ;$$

Constraint The name space of the relations with respect to their connected classes has to be unique.

$$\text{CD12}: \quad \forall\, r_1, r_2 : V_{Relation} \mid r_1.name = r_2.name \\ \bullet\, r_1 = r_2 \vee (\exists\, c : V_{ClassUnit} \bullet (c \rightharpoonup r_1) \wedge \neg\, (c \rightharpoonup r_2)) \ ;$$

❋

Containment Adornments

The *containment adornments* are further information for the has relation. The containment by value means that the life span of the component object depends on the life span of the aggregate object whereas the containment by reference means that both life spans are independent.

Both kinds of containment are represented by a square near the class of the component object. The containment by value is represented by a filled square whereas the containment by reference is represented by a hollow one.

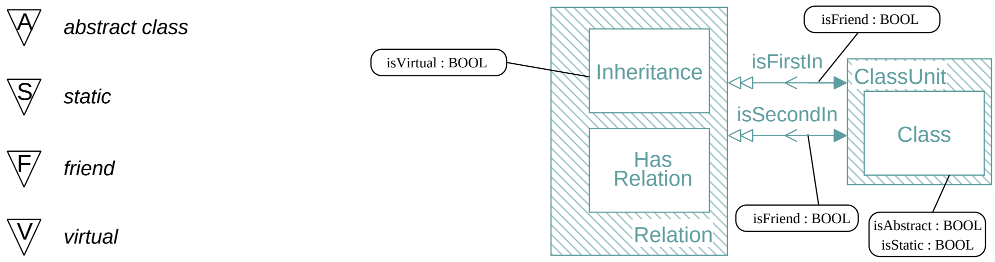
The attribute containment of the Has Relation expresses this further information in which the value 'unspecified' corresponds to the general case.

❋

Properties

Properties apply to various concepts and have the following meanings: An *abstract class* denotes a class which is not allowed to have own instances or which has an operation without an implementation. A *friend class* has a higher priority in accessing to attributes or operations of other classes connected to this class. A *virtual inheritance* is an inheritance in which the subclasses inherit the attributes of a superclass only once although there are several paths in the inheritance hierarchy to this superclass. A *static class* denotes a class which is used as a class attribute or a class operation in another class.

Properties are represented by triangle-shaped icons with the first letter of the corresponding property within it. The triangle for the abstract class is positioned inside the abstract class whereas the other triangles are positioned on the corresponding relations.

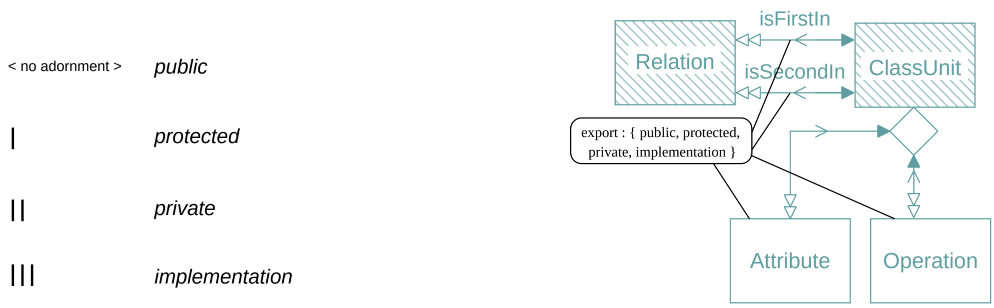


All properties are modelled by boolean attributes. A class is an Abstract class or it is not. An inheritance relation is a Virtual inheritance or it is not. A class, connected to another class via a relation, is a Friend class or it is not. And at last, if a class is connected to another class via a has relation, the class is a Static class or it is not.

Export Control

The set of object which can access attributes, operations and relations of objects of another class is restricted by export control. A *public* element is accessible to all clients whereas *protected* elements are accessible only to objects of subclasses or friend classes or objects of the class itself. A *private* element is only accessible to friends or the objects of the class itself, and *implementation* elements are inaccessible even to objects of a friend class.

Export control is represented by a number of hash marks which are positioned at a relation, an attribute or an operation.



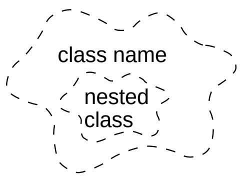
Export control is modelled by the attribute export of the concepts Attribute, Operation and Relation. The latter is done indirectly by assigning it to the isFirstIn and isSecondIn relationships.

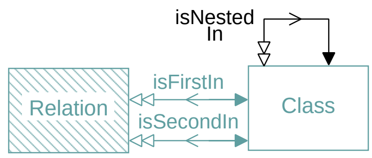
❈

Nesting

A class may be physically *nested* in another class. This has two purposes; first, it is used to control name space, and second, it encapsulates nested classes from other classes.

Nesting a class in another class is represented by positioning the cloud icon of the nested class within the cloud icon of the other class.





A class is a nested class, if the class is Nested In another class.

Constraint [7] The class name space in each category, in each nesting class and at the toplevel of a model must be unique.

$$\mathsf{CD4}: \quad \forall\, c_1, c_2 : V_{ClassUnit} \mid c_1 \neq c_2 \\ \bullet\ (c_1.name \neq c_2.name) \vee \\ (\exists\, cg : V_{Category} \bullet cg \rightharpoonup_{clusters} c_1 \wedge \neg\, (cg \rightharpoonup_{clusters} c_2)) \vee \\ (\exists\, c : V_{Class} \bullet c \leftharpoonup_{isNestedIn} c_1 \wedge \neg\, (c \leftharpoonup_{isNestedIn} c_2))\ ;$$

Constraint Nested classes are not allowed to have associations to other classes.

$$\mathsf{CD13}: \quad \forall\, c : V_{Class} \mid degree(\leftharpoonup_{isNestedIn}, c) > 0 \\ \bullet\ degree(\rightharpoonup_{isFirstIn}, c) = 0 \wedge degree(\rightharpoonup_{isSecondIn}, c) = 0\ ;$$

Constraint The classes nested in a class must be ordered in a hierarchy.

$$\mathsf{CD14}: \quad isForest(eGraph(\leftharpoonup_{isNestedIn}))\ ;$$

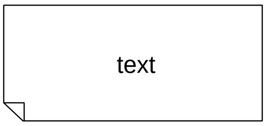
❈

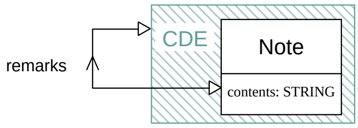
[7] We extend the **constraint** CD4 on page 11 in such a way that nesting a class has the same effect at the name space as assigning a class to a category.

Notes

Each graphical element in a class diagram can be adorned by natural language text, or so called *notes*.

A note is represented by a rectangle with a turned-up corner and the text within it. The note is connected to a certain graphical element in a class diagram by a dashed line.





Notes are assigned to elements by the remarks relationship. Elements used in a class diagram are specializations of CDE.

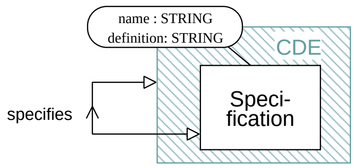
❋

2.2 Specification

A *specification* is an additional "nongraphical form used to provide the complete definition of an entity in the notation" (p. 196). Each specification has a name and a definition.

Specifications are represented by natural language text structured by keywords.

Name: identifier
Definition: text



Each element in a class diagram, expressed by CDE, may be specified by a Specification.

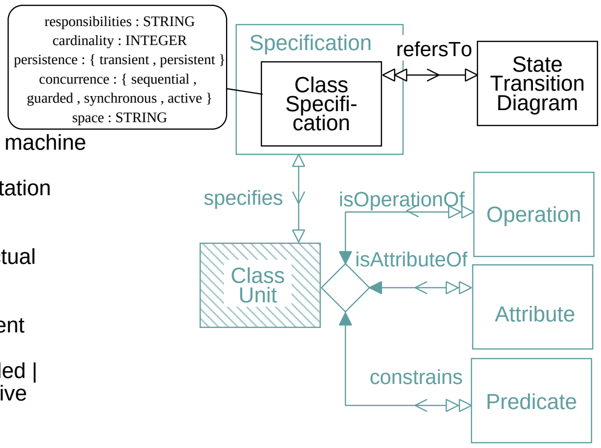
❋

Class Specification

A *class specification* contains information about a class. It contains information which is already described by the graphical notation such as attributes, operations, constraints, export controls, and parameters. And it contains information which is only described in a class specification: responsibilities, which denote responsible persons; state machine, which refers to a state transition diagram; cardinality, which expresses a limit for the number of objects; persistence, which denotes the life span; concurrence, which describes parallelism; and space complexity, which denotes required memory space.

A class specification is represented by natural language text structured by keywords.

Responsibilities:	text
Attributes:	list of attributes
Operations:	list of operations
Constraints:	list of constraints
State machine	reference to state machine
Export control:	public | implementation
Cardinality:	expression
Parameters:	list of formal or actual parameters
Persistence:	transient | persistent
Concurrence:	sequential | guarded | synchronous | active
Space complexity:	expression



A Class Specification is a Specification which specifies classes, expressed by Class Units. Attributes, Operations, Constraints, export control, and Parameters are already modelled concepts. Additionally, a State Transition Diagram can be referred by a class specification. Responsibilities, cardinality, persistence, concurrence and space complexity are modelled by attributes.

Constraint A class specification only specifies classes.

$$\text{CD15}: \quad \forall\, cs : V_{ClassSpecification} \bullet (cs \rightharpoonup_{specifies}) \subseteq V_{ClassUnit} \;;$$

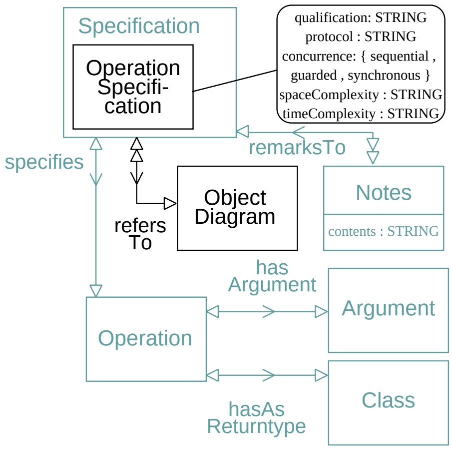
❋

Operation Specification

An *operation specification* contains information about an operation. It contains information which is already described by the graphical notation such as return class, arguments and export control. And it contains information which is only described in an operation specification: qualification, which denotes quality description; protocol, which denotes used protocols; precondition, semantics, postcondition and exceptions, which describe meanings and constraints; concurrence, which describes synchronization conditions; space and time complexity, which denotes required memory space and expected run-time behavior.

An operation specification is represented by natural language text structured by keywords.

Return class:	reference to class
Arguments:	list of formal arguments
Qualification:	text
Export control:	public | implementation | expression
Protocol:	text
Preconditions:	text | reference to source code | reference to object diagram
Semantics:	text | reference to source code | reference to object diagram
Postconditions:	text | reference to source code | reference to object diagram
Exceptions:	list of exeptions
Concurrence:	sequential | guarded | synchronous
Space complexity:	expression
Time complexity:	expression



An Operation Specification is a Specification which specifies certain Operations. The return class and arguments are already modelled by isReturnType and hasArgument relationships as well as the export control. Qualification, protocol, concurrence, space and time complexity are modelled by attributes. Precondition, semantics, postcondition and exceptions are modelled by the refersTo relationship connected to Object Diagrams, or as Notes.

Constraint An operation specification only specifies operations.

$$\text{CD16}: \quad \forall\, os : V_{OperationSpecification} \bullet (os \rightharpoonup_{specifies}) \subseteq V_{Operation} \;;$$

❄

2.3 Integration: Logical View

The EER description in figure 2 and the following collection of GRAL predicates summarize the previous ones and express the logical view of the Booch method.

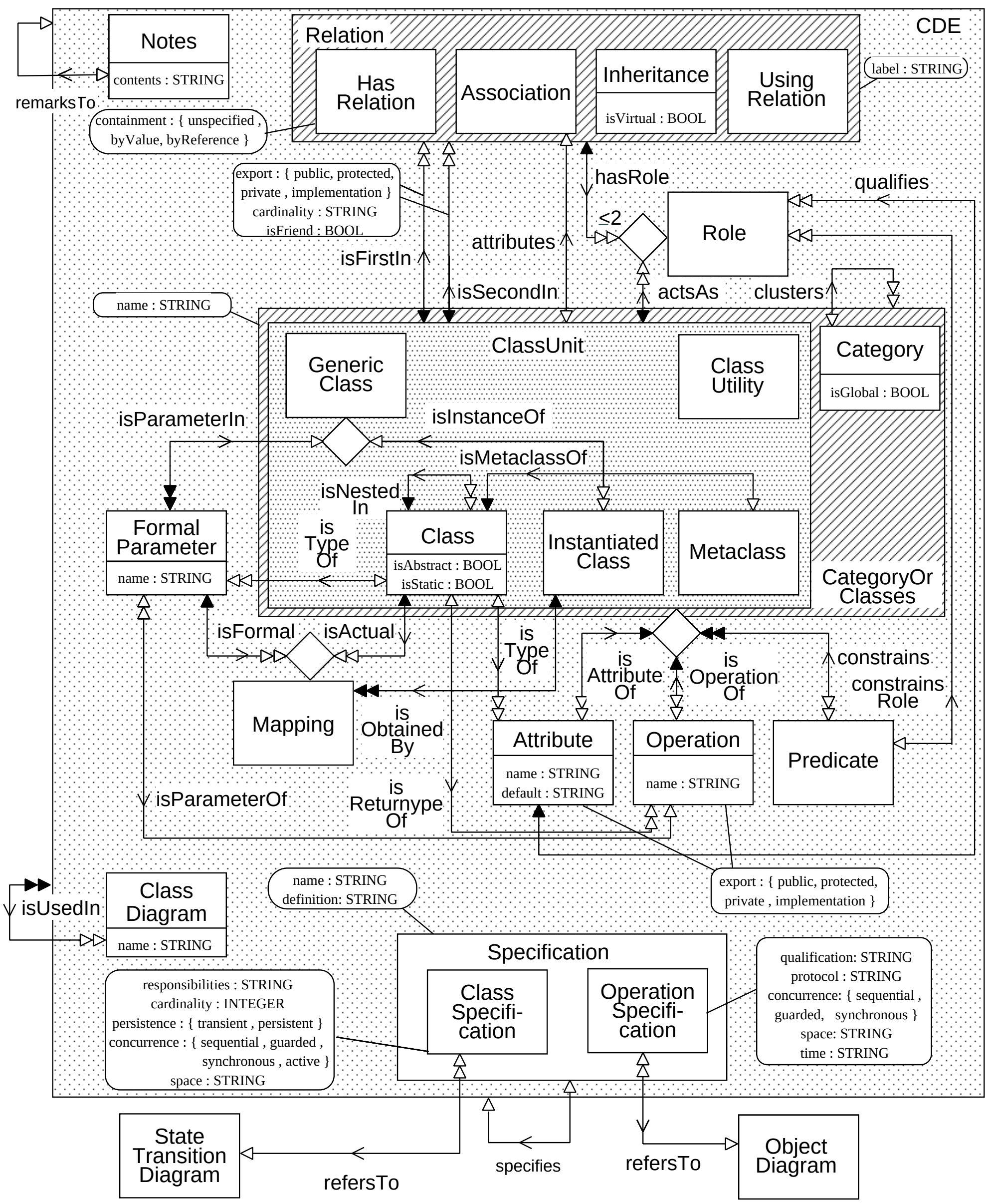


Figure 2: EER Diagram of the Logical View

forall G **in** *LogicalView* **assert**

CD1 : $\forall\, cd_1, cd_2 : V_{ClassDiagram} \mid cd_1 \neq cd_2 \bullet cd_1.name \neq cd_2.name$;

CD2 : $\forall\, cg_1, cg_2 : V_{Category} \mid cg_1 \neq cg_2 \bullet cg_1.name \neq cg_2.name$;

CD3 : $\forall\, cg : V_{Category};\ c : V_{ClassUnit} \bullet cg.name \neq c.name$;

CD4 : $\forall\, c_1, c_2 : V_{ClassUnit} \mid c_1 \neq c_2$
$\bullet\ (c_1.name \neq c_2.name) \vee$
$(\exists\, cg : V_{Category} \bullet cg \rightharpoonup_{clusters} c_1 \wedge \neg\, (cg \rightharpoonup_{clusters} c_2)) \vee$
$(\exists\, c : V_{Class} \bullet c \leftharpoonup_{isNestedIn} c_1 \wedge \neg\, (c \leftharpoonup_{isNestedIn} c_2))$;

CD5 : $isForest(eGraph(\rightharpoonup_{clusters}))$;

CD6 : $\forall\, ic : V_{InstantiatedClass};\ gc : V_{GenericClass} \mid ic \rightharpoonup_{isInstanceOf} gc$
$\bullet\ (\forall\, fp : V_{FormalParameter} \mid fp \rightharpoonup_{isParameterIn} gc$
$\bullet\ fp \rightharpoonup_{isFormal} \leftharpoonup_{isObtainedBy} ic\ )$;

CD7 : $\forall\, cg : V_{Category} \bullet (cg \rightharpoonup_{isFirstIn} \cup\, cg \rightharpoonup_{isSecondIn}) \subseteq V_{UsingRelation}$;

CD8 : $\forall\, c_1, c_2 : V_{ClassUnit} \mid c_1 \rightharpoonup_{isFirstIn} \leftharpoonup_{isSecondIn} c_2$
$\bullet\ (degree(\leftharpoonup_{clusters}, c_1) = 0 \wedge degree(\leftharpoonup_{clusters}, c_2) = 0) \vee$
$(\exists\, cg : V_{Category} \bullet cg(\rightharpoonup_{clusters})^{+} c_2 \wedge cg.isGlobal = TRUE) \vee$
$(\exists\, cg : V_{Category} \bullet cg(\rightharpoonup_{clusters})^{+} c_1 \leftharpoonup_{clusters} (\rightharpoonup_{clusters})^{+} c_2) \vee$
$(c_1(\leftharpoonup_{clusters})^{+} \rightharpoonup_{isFirstIn} \bullet_{UsingRelation} \leftharpoonup_{isSecondIn} (\rightharpoonup_{clusters})^{+} c_2)$;

CD9 : $\forall\, v : V_{ClassUnit} \bullet \neg\, (c\ (\rightharpoonup_{isFirstIn} \bullet_{Inheritance} \leftharpoonup_{isSecondIn})^{+} c)$;

CD10 : $\forall\, ic : V_{InstantiatedClass}$
$\bullet\ (\forall\, m : V_{Mapping} \mid ic \rightharpoonup_{isObtainedBy} m$
$\bullet\ (m \leftharpoonup_{isActual} (\rightharpoonup \bullet_{Association} \leftharpoonup \mid$
$\rightharpoonup_{isSecondIn} \bullet_{UsingRelation} \leftharpoonup_{isFirstIn})\ ic))$;

CD11 : $\forall\, ro : V_{Role};\ a : V_{Attribute};\ c : V_{ClassUnit} \mid a \rightharpoonup_{qualifies} ro \leftharpoonup_{actsAs} c$
$\bullet\ (\exists\, c_1 : V_{ClassUnit};\ r : V_{Relation} \mid r \rightharpoonup_{hasRole} ro$
$\bullet\ (c \rightharpoonup_{isFirstIn} r \leftharpoonup_{isSecondIn} c_1 \leftharpoonup_{isAttributeOf} a) \vee$
$(c \rightharpoonup_{isSecondIn} r \leftharpoonup_{isFirstIn} c_1 \leftharpoonup_{isAttributeOf} a))$;

CD12 : $\forall\, r_1, r_2 : V_{Relation} \mid r_1.name = r_2.name$
$\bullet\; r_1 = r_2 \lor (\exists\, c : V_{ClassUnit} \bullet (c \rightharpoonup r_1) \land \neg\,(c \rightharpoonup r_2))$;

CD13 : $\forall\, c : V_{Class} \mid degree(\leftharpoonup_{isNestedIn}, c) > 0$
$\bullet\; degree(\rightharpoonup_{isFirstIn}, c) = 0 \land degree(\rightharpoonup_{isSecondIn}, c) = 0$;

CD14 : $isForest(eGraph(\leftharpoonup_{isNestedIn}))$;

CD15 : $\forall\, cs : V_{ClassSpecification} \bullet (cs \rightharpoonup_{specifies}) \subseteq V_{ClassUnit}$;

CD16 : $\forall\, os : V_{OperationSpecification} \bullet (os \rightharpoonup_{specifies}) \subseteq V_{Operation}$;

3 Dynamic View

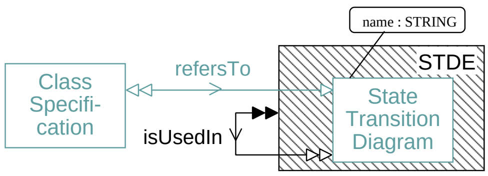
The *dynamic view* of a system is described by the order of sequences of operations which are possible in this system and is represented by two sets of documents:

- a set of *state transition diagrams*, and
- a set of *object diagrams*.

3.1 State Transition Diagram

A *state transition diagram* is a named sheet of paper showing one or more icons which represent the state space of a given class, the events that cause a transition from one state to another, and the actions which are triggered by these events.

A state transition diagram has no explicit icon. One may take the sheet of paper as the graphical representation.



A State Transition Diagram comprises elements, which are assigned to it by the isUsedIn relationship. Analogous to the class diagram elements in the logical view the STDE is used as a placeholder for all concepts which can be assigned to a state transition diagram. For graphical simplicity, this is done in figure 3 (p. 35) at first. A state transition diagram refers To zero, one or more Class Specifications which denote classes. In case of no reference the diagram refers to the system as a whole.

Constraint The name space of state transition diagrams has to be unique in a system.

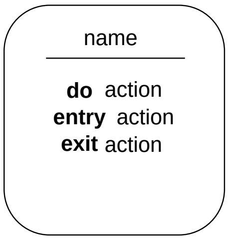
$$\text{STD1}: \quad \forall\, std_1, std_2 : V_{StateTransitionDiagram} \mid std_1 \neq std_2 \bullet std_1.name \neq std_2.name \;;$$

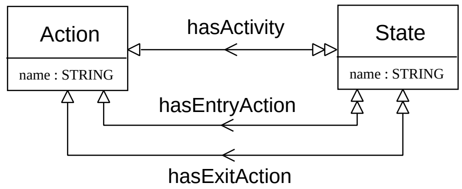
❋

State Icon

A *state* of an object denotes a period of time in which the object does not change its properties connected to this state. During this period an action can be executed, and at the beginning and at the end an entry- respectively an exit-action may occur.

A state is represented by a rounded rectangle labeled by its name. The actions are listed below a line within the rectangle distinguished by the keywords ‘do’, ‘entry’ and ‘exit’





A State can be connected to an Action in the three different manners described above: the hasActivity denotes the do-action, the hasEntryAction denotes the entry-action, and the hasExitAction denotes the exit-action.

Constraint The name space of states has to be unique in a system.

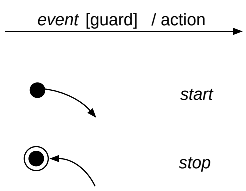
$$\mathsf{STD2}: \quad \forall\, s_1, s_2 : V_{State} \mid s_1 \neq s_2 \bullet s_1.name \neq s_2.name \ ;$$

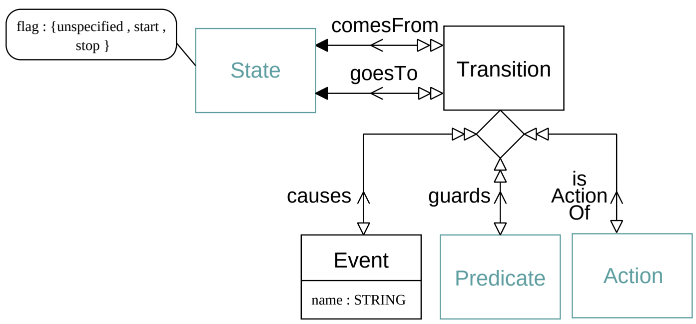
❋

State Transitions

An *event* is a stimulus that causes a *transition* from one state to another and triggers an *action*. An event may have a *guard* which is a semantic condition that must be valid additionally. A start state denotes the creation of an object whereas a stop state denotes a deletion.

A transition is represented by a line with an arrow connecting two states. Events, guards and actions are annotated as text. A start state is represented by a bullet and an arrow pointing to this state, the stop state is represented similar but with an additional circle around the bullet.





A Transition exactly starts at one state, expressed by the comesFrom relationship, and exactly ends at one state, expressed by the goesTo relationship. An Event causes this transition, a Predicate guards it and an Action is Triggered By that event. Start and stop states are recognizable by the value of the attribute flag which may be start, stop or unspecified.

Constraint The name space for each event of the outgoing transitions of a state must be unique.

$$\text{STD3}: \quad \forall\, e_1, e_2 : V_{Event} \mid e_1.name = e_2.name \bullet e_1 = e_2 \lor \neg\, (e_1 \rightharpoonup_{causes} \rightharpoonup_{comesFrom} \leftharpoonup_{comesFrom} \leftharpoonup_{causes} e_2)\ ;$$

Constraint A stop state is not allowed to have outgoing transitions.

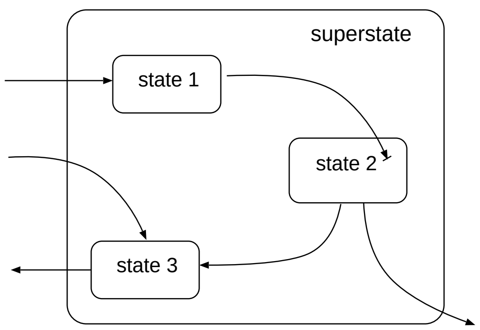
$$\text{STD4}: \quad \forall\, s : V_{State} \bullet s.flag = stop \land degree(\leftharpoonup_{comesFrom}, s) = 0\ ;$$

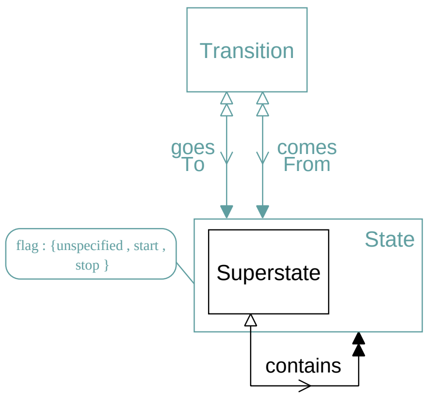
❋

Nesting

A superstate is a state which contains one or more other states, called *nested* states.

A superstate is represented by a state icon in which the nested states are drawn within this state icon.





A Superstate may contain one or more other States.

There are several constraints connected to superstates.

Constraint In a superstate may exist only one direct start state.

$$\text{STD5}: \quad \forall\, su : V_{Superstate} \bullet \{s : V_{State} \mid su \rightharpoonup_{contains} s \land s.flag = start\} \leq 1\ ;$$

Constraint If a superstate has a direct transition going to it, one of its direct nested states must be a start state.

$$\text{STD6}: \quad \forall\, su : V_{Superstate} \mid degree(\leftharpoonup_{goesTo}, su) > 0 \bullet (\exists_1\, s : V_{State} \bullet su \rightharpoonup_{contains} s \land s.flag = start\ )\ ;$$

Constraint The states nested in a category must be ordered in a hierarchy.

STD7 : $isForest(eGraph(\rightharpoonup_{contains}))$;

Constraint On the top of the diagram must exist one start state.

STD8 : $\forall\, std : V_{StateTransitionDiagram}$
$\bullet\ (\exists_1\, s : V_{State} \bullet s \rightharpoonup_{isUsedIn} std \land \text{degree}(\leftharpoonup_{contains}, s) = 0 \land s.flag = start)$;

History

A superstate with a history describes the semantics that one return to the most recently visited substate when transitioning directly to this superstate.

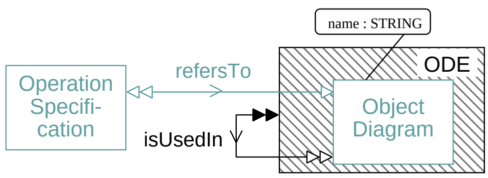
A superstate with a history is represented by a state icon and a small circle with the letter 'H' placed within this state icon.

The attribute hasHistory of a Superstate denotes whether a superstate has a history or has not.

3.2 Object Diagram

An *object diagram* is a named sheet of paper showing one or more icons which represent certain configuration of objects and a possible order of set messages.

An object diagram has no explicit icon. It is graphically represented by using a certain sheet of paper.



An Object Diagram comprises elements, which are assigned to it by the isUsedIn relationship. Analogous to the class diagram element in the logical view the ODE is used as a placeholder for all concepts which can be assigned to a Object Diagram. For graphical simplicity, this is done in figure 3 (p. 35) at first. An Object Diagram refers To zero, one or more Class Specification which denotes classes.

Constraint The name space of object diagrams has to be unique in a system.

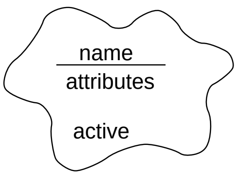
$$\text{OD1}: \quad \forall\, od_1, od_2 : V_{ObjectDiagram} \mid od_1 \neq od_2 \bullet od_1.name \neq od_2.name \ ;$$

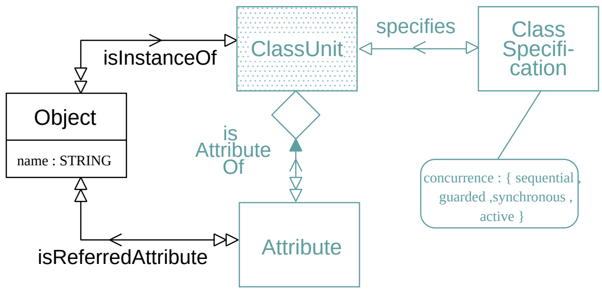
❄

Object Icon

An *object* is an instance of a class. In the Booch method an active object embodies its own thread of control. Thus, it can change its state independently of messages of other objects.

An object is represented by a cloud icon. Within this cloud the name, possibly attributes and the word 'active' are listed one after another.





An Object is an Instance Of a class, which is expressed by Class Unit. If an attribute is explicitly referred in an object diagram, this is modelled by the isReferredAttribute relationship. An active object can be identified by the value of the attribute concurrence in its Class Specification.

Constraint The attributes, which are referred by an object, must be attributes of its class.

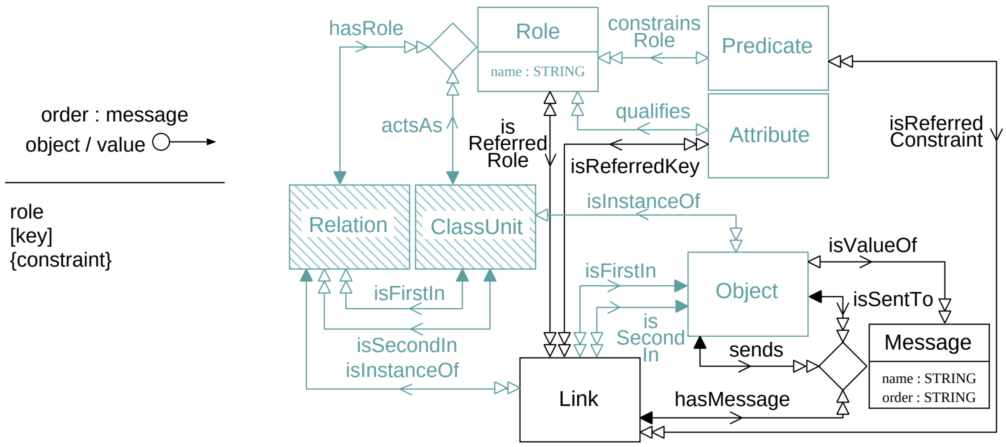
$$\text{OD2}: \quad \forall\, o : V_{Object};\ a : V_{Attribute} \mid o \leftharpoondown_{isReferredAttribute} a \bullet o \rightharpoonup_{isInstanceOf} \leftharpoondown_{isAttributeOf} a \ ;$$

❄

Link

A *link* is a physical or logical connection between two objects which is required for message passing between these objects. The messages can be ordered and can possess objects or values.

A link is represented by a line connecting two objects. The ordering is represented by numbers. The name of the message, object and value are annotated above the line. Additionally, roles, keys and constraints can be referred from the class diagram.



A Link is connected to two Objects by the isFirstIn and isSecondIn relationships, and it is an Instance Of a Relation. A Message is sent from one object to another object via a link, which is expressed by the sends, isSentTo and hasMessage relationships. Additionally, an object is a value Of this message. Roles, attributes and constraints can be referred from the class diagram by the isReferredRole, isReferredKey and isReferredConstraint relationships.

Constraint The roles, which are referred by a link, must be roles of its relation.

$$\text{OD3}: \quad \forall\, l : V_{Link};\ ro : V_{Role} \mid l \leftharpoonup_{isReferredRole} ro \\ \bullet\ l \rightharpoonup_{isInstanceOf} \rightharpoonup_{hasRole} ro\ ;$$

Constraint The keys, which are referred by a link, must be keys of its relation.

$$\text{OD4}: \quad \forall\, l : V_{Link};\ a : V_{Attribute} \mid l \leftharpoonup_{isReferredKey} a \\ \bullet\ l \rightharpoonup_{isInstanceOf} \rightharpoonup_{hasRole} \leftharpoonup_{qualifies} a\ ;$$

Constraint The constraints, which are referred by a link, must be constraints of its relation.

$$\text{OD5}: \quad \forall\, l : V_{Link};\ p : V_{Predicate} \mid l \leftharpoonup_{isReferredConstraint} p \\ \bullet\ l \rightharpoonup_{isInstanceOf} \rightharpoonup_{hasRole} \leftharpoonup_{constrainsRole} p\ ;$$

Constraint The links between two objects must be instances of a relation between the classes of the objects.

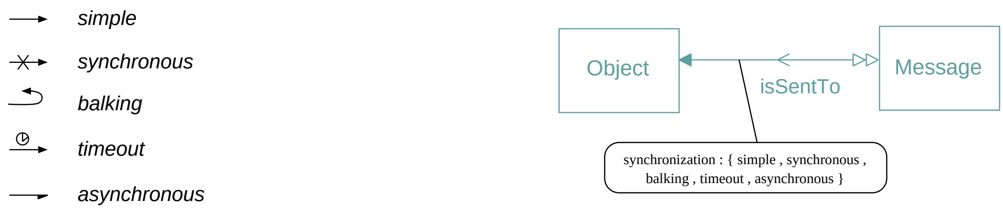
$$\text{OD6}: \quad \forall\, l : V_{Link};\ o_1, o_2 : V_{Object} \mid l \rightharpoonup_{isFirstIn} o_1 \wedge l \rightharpoonup_{isSecondIn} o_2 \\ \bullet\ (\exists\, r : V_{Relation} \mid l \rightharpoonup_{isInstanceOf} r \\ \bullet\ (o_1 \rightharpoonup_{isInstanceOf} \rightharpoonup_{isFirstIn} r \leftharpoonup_{isSecondIn} \leftharpoonup_{isInstanceOf} o_2) \vee \\ (o_1 \rightharpoonup_{isInstanceOf} \rightharpoonup_{isSecondIn} r \leftharpoonup_{isFirstIn} \leftharpoonup_{isInstanceOf} o_2))\ ;$$

❄

Synchronization

Synchronization describes conditions on message passing between two active objects. *Simple synchronization* requires no conditions. In *synchronous synchronization* the message sending object must wait until the receiving object is ready, in *balking synchronization* the message passing fails if the receiving object is not ready, *timeout synchronization* defines a period for sucessful message passing and *asynchronous synchronization* means that the sending object gets no confirmation of its message.

The different kinds of synchronization are represented by arrows with additional annotations above.



Synchronization is modelled by the attribute synchronization in which the value determines the kind of synchronization. It is an attribute of the isSentTo relationship which refers to the receiving object in a message.

⁂

Visibility

Visibility describes "how instances can see one another" (p. 213). An object can see another object as *global*, as a *parameter* of an operation, as a *part* of the client object, or as a *locally* declared one. Additionally, an object can be shared by several other objects.

Visibility is represented by a letter within a rectangle positioned between object and connecting link. Sharing is represented by filled rectangles.

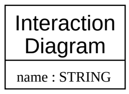
Visibility is modelled by the attribute visibility of the isFirstIn and isSecondIn relationships which express the connecting between Objects and Links. Sharing is expressed by the boolean attribute isShared.

⁂

3.3 Interaction Diagram

An *interaction diagram* is another graphical representation of an object diagram in which the messages ordered by their numbers are listed from top to bottom.

A interaction diagram has no explicit icon. One may take the sheet of paper as the graphical representation.

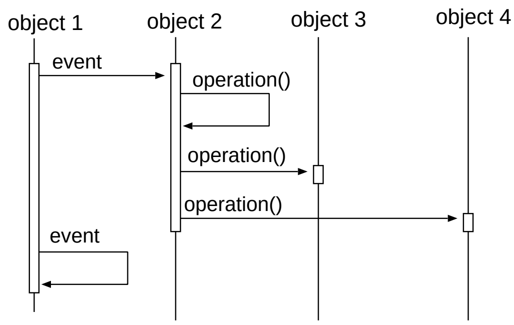


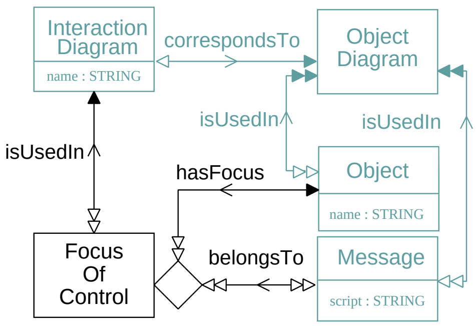
A interaction diagram is modelled by the concept Interaction Diagram.

❋

Only one information is added to an interaction diagram: the *focus of control* which describes what operations belong together depending on occurring messages.

In an interaction diagram objects are represented by vertical lines. Messages are represented by horizontal arrows and focus of control is represented by a rectangle.





The Focus of Control is assigned to an Interaction Diagram by the isUsedBy relationship. An Interaction Diagram always corresponds To an Object Diagram. It may add a Focus Of Control to an Object and a number of Messages which belong To the focus.

Constraint The interaction diagram may only describe additional information on such elements which belong to its corresponding object diagram.

$$\text{OD7}: \quad \forall\, id : V_{InteractionDiagram};\ foc : V_{FocusOfControl} \mid id \leftharpoonup_{isUsedIn} foc \\ \bullet\ (\forall\, o : V_{Object} \mid o \rightharpoonup_{hasFocus} foc \\ \bullet\ o \rightharpoonup_{isUsedIn} \leftharpoonup_{correspondsTo} id) \wedge \\ (\forall\, m : V_{Message} \mid m \rightharpoonup_{belongsTo} foc \\ \bullet\ m \rightharpoonup_{isUsedIn} \leftharpoonup_{correspondsTo} id)\ ;$$

3.4 Integration: Dynamic View

The EER description in figure 3 and the following collection of GRAL predicates summarize the previous ones and express the dynamic view of the Booch method. Furthermore, two constraints are added.

Constraint An event which is mapped to an operation, must belong to a state transition diagram which is used for specifying the class of the operation.

STD9 : $\forall\, e : V_{Event};\ o : V_{Operation} \mid e \rightharpoonup_{mapsTo} o$
$\bullet\ e \rightharpoonup_{isUsedIn} \leftharpoonup_{refersTo} \rightharpoonup_{specifies} \leftharpoonup_{isOperationOf}\ o\ ;$

Constraint An action must either be mapped to an operation or it must trigger another event.

STD10 : $\forall\, a : V_{Action} \bullet degree(\rightharpoonup_{mapsTo}, a) + degree(\rightharpoonup_{triggers}, a) = 1\ ;$

❋

forall G **in** *DynamicView* **assert**

STD1 : $\forall\, std_1, std_2 : V_{StateTransitionDiagram} \mid std_1 \neq std_2$
$\bullet\ std_1.name \neq std_2.name\ ;$

STD2 : $\forall\, s_1, s_2 : V_{State} \mid s_1 \neq s_2 \bullet s_1.name \neq s_2.name\ ;$

STD3 : $\forall\, e_1, e_2 : V_{Event} \mid e_1.name = e_2.name$
$\bullet\ e_1 = e_2 \vee \neg\,(e_1 \rightharpoonup_{causes} \rightharpoonup_{comesFrom} \leftharpoonup_{comesFrom} \leftharpoonup_{causes} e_2)\ ;$

STD4 : $\forall\, s : V_{State} \bullet s.flag = stop \wedge degree(\leftharpoonup_{comesFrom}, s) = 0\ ;$

STD5 : $\forall\, su : V_{Superstate}$
$\bullet\ \{s : V_{State} \mid su \rightharpoonup_{contains} s \wedge s.flag = start\} \leq 1\ ;$

STD6 : $\forall\, su : V_{Superstate} \mid degree(\leftharpoonup_{goesTo}, su) > 0$
$\bullet\ (\exists_1\, s : V_{State} \bullet su \rightharpoonup_{contains} s \wedge s.flag = start\ )\ ;$

STD7 : $isForest(eGraph(\rightharpoonup_{contains}))\ ;$

STD8 : $\forall\, std : V_{StateTransitionDiagram}$
$\bullet\ (\exists_1\, s : V_{State} \bullet s \rightharpoonup_{isUsedIn} std \wedge \mathrm{degree}(\leftharpoonup_{contains}, s) = 0 \wedge$
$s.flag = start)\ ;$

STD9 : $\forall\, e : V_{Event};\ o : V_{Operation} \mid e \rightharpoonup_{mapsTo}\ o$
$\bullet\ e \rightharpoonup_{isUsedIn} \leftharpoonup_{refersTo} \rightharpoonup_{specifies} \leftharpoonup_{isOperationOf}\ o\ ;$

STD10 : $\forall\, a : V_{Action} \bullet degree(\rightharpoonup_{mapsTo}, a) + degree(\rightharpoonup_{triggers}, a) = 1\ ;$

OD1 : $\forall\, od_1, od_2 : V_{ObjectDiagram} \mid od_1 \neq od_2 \bullet od_1.name \neq od_2.name\ ;$

OD2 : $\forall\, o : V_{Object};\ a : V_{Attribute} \mid o \leftharpoonup_{isReferredAttribute}\ a$
$\bullet\ o \rightharpoonup_{isInstanceOf} \leftharpoonup_{isAttributeOf}\ a\ ;$

OD3 : $\forall\, l : V_{Link};\ ro : V_{Role} \mid l \leftharpoonup_{isReferredRole}\ ro$
$\bullet\ l \rightharpoonup_{isInstanceOf} \rightharpoonup_{hasRole}\ ro\ ;$

OD4 : $\forall\, l : V_{Link};\ a : V_{Attribute} \mid l \leftharpoonup_{isReferredKey}\ a$
$\bullet\ l \rightharpoonup_{isInstanceOf} \rightharpoonup_{hasRole} \leftharpoonup_{qualifies}\ a\ ;$

OD5 : $\forall\, l : V_{Link};\ p : V_{Predicate} \mid l \leftharpoonup_{isReferredConstraint}\ p$
$\bullet\ l \rightharpoonup_{isInstanceOf} \rightharpoonup_{hasRole} \leftharpoonup_{constrainsRole}\ p\ ;$

OD6 : $\forall\, l : V_{Link};\ o_1, o_2 : V_{Object} \mid l \rightharpoonup_{isFirstIn}\ o_1 \wedge l \rightharpoonup_{isSecondIn}\ o_2$
$\bullet\ (\exists\, r : V_{Relation} \mid l \rightharpoonup_{isInstanceOf}\ r$
$\bullet\ (o_1 \rightharpoonup_{isInstanceOf} \rightharpoonup_{isFirstIn}\ r \leftharpoonup_{isSecondIn} \leftharpoonup_{isInstanceOf}\ o_2) \vee$
$(o_1 \rightharpoonup_{isInstanceOf} \rightharpoonup_{isSecondIn}\ r \leftharpoonup_{isFirstIn} \leftharpoonup_{isInstanceOf}\ o_2))\ ;$

OD7 : $\forall\, id : V_{InteractionDiagram};\ foc : V_{FocusOfControl} \mid id \leftharpoonup_{isUsedIn}\ foc$
$\bullet\ (\forall\, o : V_{Object} \mid o \rightharpoonup_{hasFocus}\ foc$
$\bullet\ o \rightharpoonup_{isUsedIn} \leftharpoonup_{correspondsTo}\ id) \wedge$
$(\forall\, m : V_{Message} \mid m \rightharpoonup_{belongsTo}\ foc$
$\bullet\ m \rightharpoonup_{isUsedIn} \leftharpoonup_{correspondsTo}\ id)\ ;$

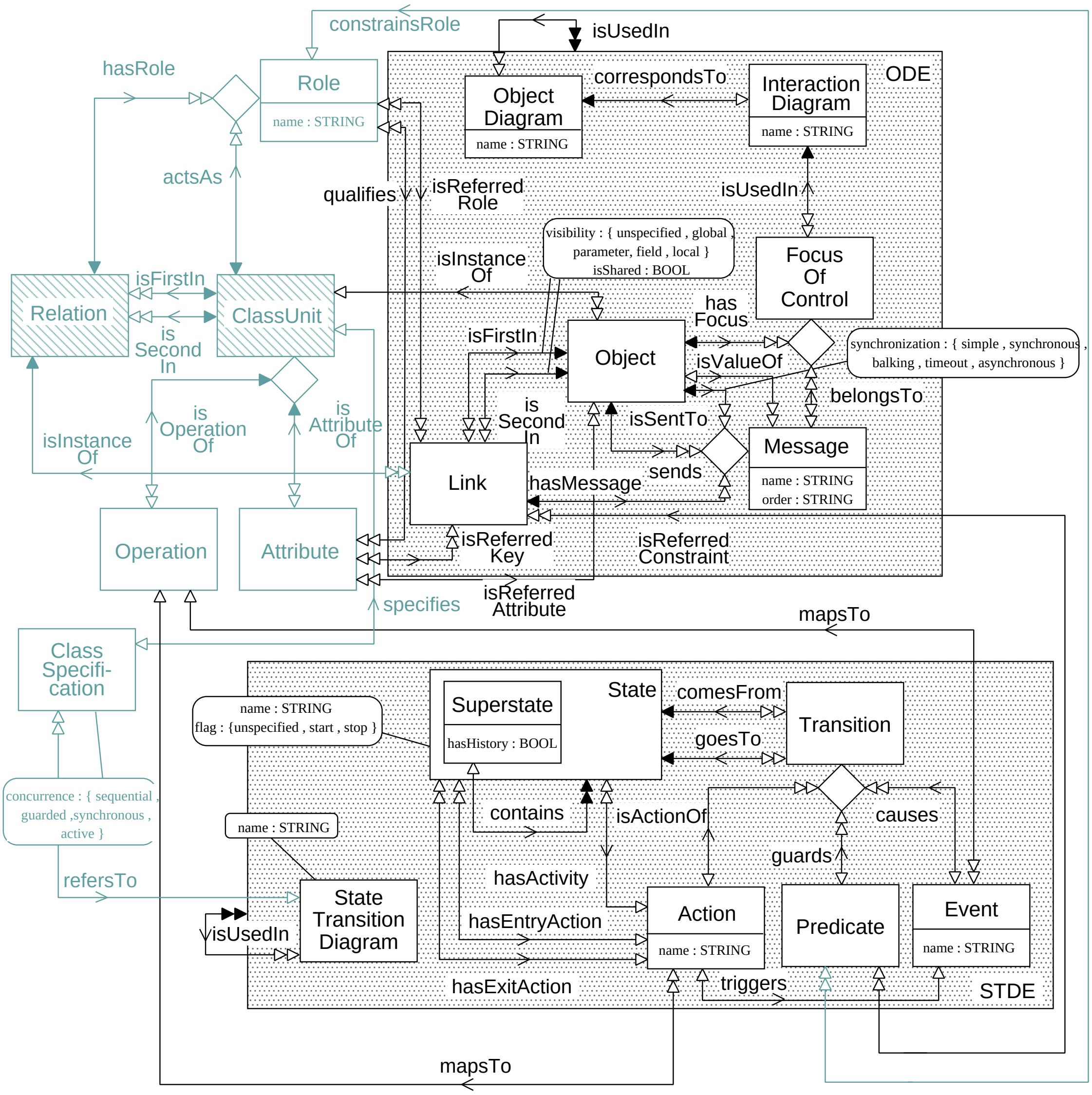


Figure 3: EER Diagram of the Dynamic View

4 Physical View

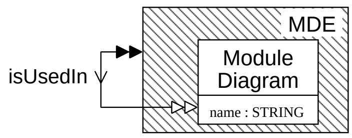
The *physical view* of a system is described by the physical allocation of its classes to subsystems and processors and is represented by two sets of documents:

- a set of *module diagrams*, and
- a set of *process diagrams*.

4.1 Module Diagram

A *module diagram* is a named sheet of paper showing one or more icons which represent the physical allocation of the source code of the system and the classes, respectively.

A module diagram has no corresponding icon. One may take the sheet of paper as the graphical representation.



A Module Diagram comprises elements which are assigned to it by the isUsedIn relationship. The abstract concept module diagram element, MDE for short, is used as a placeholder for all these elements which can be assigned to a module diagram. I.e. all these elements has to be specialized concepts of the MDE. For graphical simplicity, this is done in figure 4 (p. 40) at first.

Constraint Each module diagram name in a system has to be distinct from all other module diagram names.

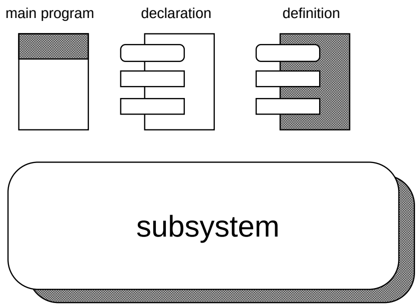
$$\text{MD1}: \quad \forall\, md_1, md_2 : V_{ModuleDiagram} \mid md_1 \neq md_2 \bullet md_1.name \neq md_2.name \;;$$

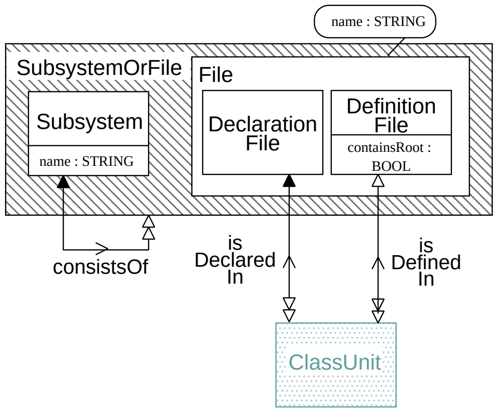
❋

Module Icons

In the Booch method a *module* denotes a file containing the declarations or the definitions of some classes. Additionally, a module can be assigned to a subsystem, which is a collection of modules, and it can contain a root class or a main program, respectively.

Declaration and definition modules are represented by hollow respectively filled rectangles with smaller rectangles placed on the left side. Definition modules which contain a root class are represented by a rectangle with another filled rectangle inside. Subsystems are represented by rounded rectangles, an additional shadow and the name within the icon.





Declaration and definition modules are expressed by Declaration Files and Definition Files, and can be assigned to Subsystems by the consistsOf relationship. A definition file can contain a Root class. Classes respectively Class Units are assigned to files by the isDefinedIn and the isDeclaredIn relationships. The concept File is used to make the model simpler.

Constraint Each subsystem name in a system has to be unique.

$$\text{MD2}: \quad \forall\, sb_1, sb_2 : V_{Subsystem} \mid sb_1 \neq sb_2 \bullet sb_1.name \neq sb_2.name \; ;$$

Constraint Each declaration respectively definition file name in a system – with respect to each subsystem and with respect to the toplevel – has to be unique.

$$\begin{aligned}\text{MD3}: \quad & \forall\, f_1, f_2 : V_{File} \mid f_1 \neq f_2 \\ & \bullet f_1.name \neq f_2.name \lor \\ & \quad (type(f_1) = DeclarationFile \land type(f_2) = DefinitionFile) \lor \\ & \quad (\exists\, sb : V_{Subsystem} \bullet (sb \rightharpoonup_{consistsOf} f_1) \land \neg\, (sb \rightharpoonup_{consistsOf} f_2) \; ;\end{aligned}$$

Constraint In each subsystem only one direct main program is allowed at least.

$$\begin{aligned}\text{MD4}: \quad & \forall\, sb : V_{Subsystem} \\ & \bullet \{f : V_{DefinitionFile} \mid \\ & \quad f.containsRoot = TRUE \land sb \rightharpoonup_{consistsOf} f\} \leq 1 \; ;\end{aligned}$$

Constraint In each system one direct main program must exist.

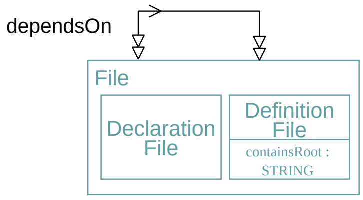
$$\begin{aligned}\text{MD5}: \quad & \forall\, md : V_{ModuleDiagram} \\ & \bullet (\exists_1 f : V_{DefinitionFile} \mid \\ & \quad f.containsRoot = TRUE \land degree(\leftharpoondown_{consistsOf}, f) = 0) \; ;\end{aligned}$$

❄

Dependency

In the Booch method a dependency between two files is a compilation dependency between these files.

A compilation dependency is represented by a line with an arrow pointing to the file on which the dependency exist.



A File depends On the changes of one or more other Files which can be Definition Files or a Declaration Files.

Constraint Circles are not allowed in the dependencies among files.

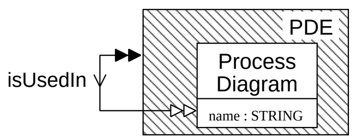
MD6 : $\forall f : V_{File} \bullet \neg (f (\rightharpoonup_{dependsOn})^{+} f)$;

❋

4.2 Process Diagram

A *process diagram* is a named sheet of paper showing one or more icons which represent the physical allocation of processes to their executing processors.

A process diagram has no explicit icon. One may take the sheet of paper as the graphical representation.



A Process Diagram comprises elements which are assigned to it by the isUsedIn relationship. The abstract concept process diagram element, PDE for short, is used as a placeholder for all these elements which can be assigned to a process diagram. I.e. all these elements has to be specialized concepts of the PDE. For graphical simplicity, this is done in figure 4 (p. 40) at first.

Constraint Each process diagram name in a system has to be unique.

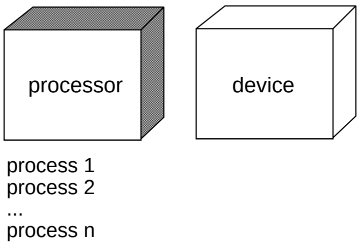
PD1 : $\forall pd_1, pd_2 : V_{ProcessDiagram} \mid pd_1 \neq pd_2$
$\bullet\ pd_1.name \neq pd_2.name$;

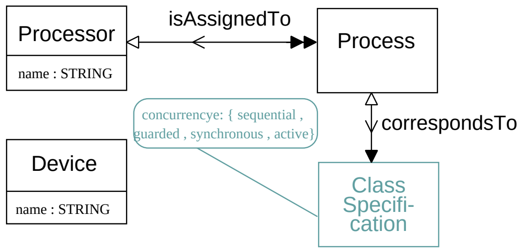
❋

Icons

Processors and *devices* denote pieces of hardware in a system. A processor is capable of executing processes in contrast to a device which is incapable of executing programs.

Processors and devices are similarly represented by cubes but a cube for a processor has filled sides. The processes, which are executed by a processor, are listed below its cube.





Processors and devices are modelled by Processors and Devices. A Process has to be assigned To a processor and it must correspond To a Class Specification.

Constraint Each class specification which has an active concurrency must have a corresponding process.

$$\mathsf{PD2}: \quad \forall\, cs : V_{ClassSpecification} \mid cs.concurrence = active \\ \bullet\ \exists\, p : V_{Process} \bullet p \rightharpoonup_{correspondsTo} cs\ ;$$

Constraint Each process must correspond to a class specification which describes the concurrence as active.

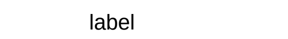
$$\mathsf{PD3}: \quad \forall\, p : V_{Process} \\ \bullet\ (\exists_1\, cs : V_{ClassSpecification} \\ \bullet\ p \rightharpoonup_{correspondsTo} cs \wedge cs.concurrence = active)\ ;$$

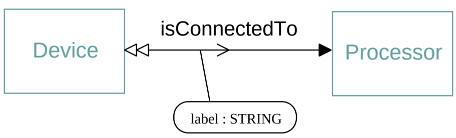
❄

Connection

A connection between a processor and a device denotes a hardware connection between them.

A connection is represented by a line which can be labeled by a name.





A connection is modelled by the isConnectedTo relationship.

4.3 Integration: Physical View

The EER description in figure 4 and the following collection of GRAL predicates summarize the previous ones and express the physical view of the Booch method.

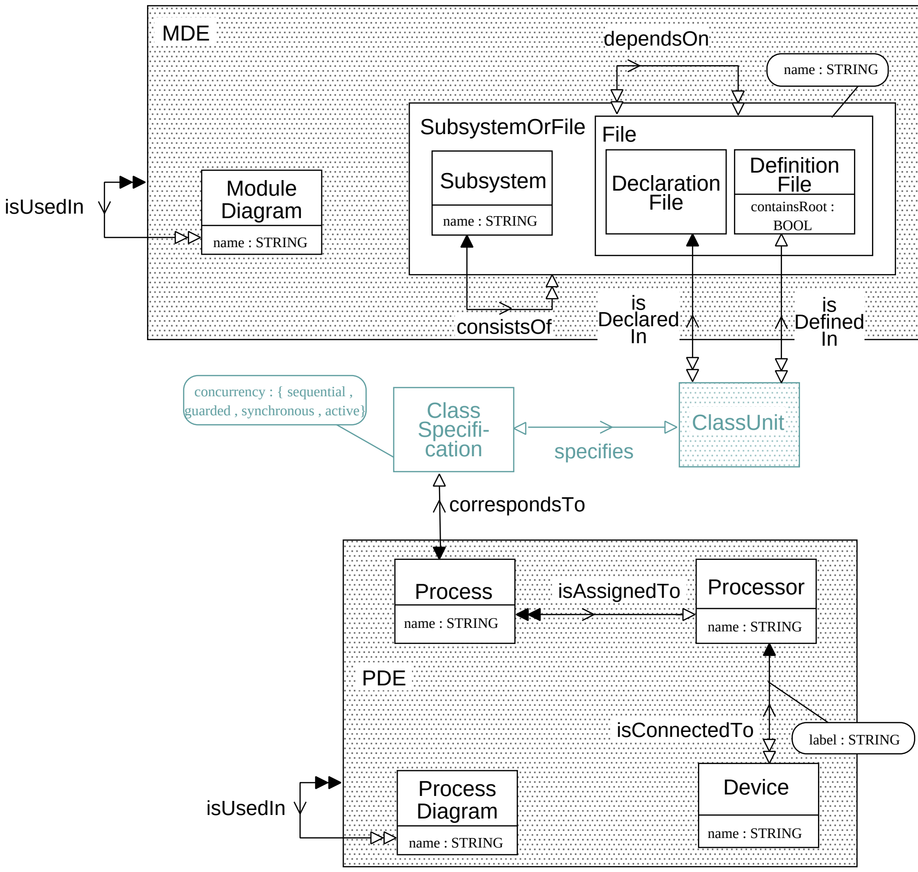


Figure 4: EER Diagram of the Physical View

forall G **in** *PhysicalView* **assert**

MD1 : $\forall\, md_1, md_2 : V_{ModuleDiagram} \mid md_1 \neq md_2$
$\bullet\; md_1.name \neq md_2.name$;

MD2 : $\forall\, sb_1, sb_2 : V_{Subsystem} \mid sb_1 \neq sb_2 \bullet sb_1.name \neq sb_2.name$;

MD3 : $\forall\, f_1, f_2 : V_{File} \mid f_1 \neq f_2$
$\bullet\; f_1.name \neq f_2.name \vee$
$(type(f_1) = DeclarationFile \wedge type(f_2) = DefinitionFile) \vee$
$(\exists\, sb : V_{Subsystem} \bullet (sb \rightharpoonup_{consistsOf} f_1) \wedge \neg\, (sb \rightharpoonup_{consistsOf} f_2)$;

MD4 : $\forall\, sb : V_{Subsystem}$
$\bullet\ \{f : V_{DefinitionFile} \mid$
$f.containsRoot = TRUE \wedge sb \rightharpoonup_{consistsOf} f\} \leq 1$;

MD5 : $\forall\, md : V_{ModuleDiagram}$
$\bullet\ (\exists_1 f : V_{DefinitionFile} \mid$
$f.containsRoot = TRUE \wedge degree(\leftharpoonup_{consistsOf}, f) = 0)$;

MD6 : $\forall f : V_{File} \bullet \neg\, (f\,(\rightharpoonup_{dependsOn})^{+} f)$;

PD1 : $\forall\, pd_1, pd_2 : V_{ProcessDiagram} \mid pd_1 \neq pd_2$
$\bullet\ pd_1.name \neq pd_2.name$;

PD2 : $\forall\, cs : V_{ClassSpecification} \mid cs.concurrence = active$
$\bullet\ \exists\, p : V_{Process} \bullet p \rightharpoonup_{correspondsTo} cs$;

PD3 : $\forall\, p : V_{Process}$
$\bullet\ (\exists_1\, cs : V_{ClassSpecification}$
$\bullet\ p \rightharpoonup_{correspondsTo} cs \wedge cs.concurrence = active)$;

References

[B94] Booch, Grady; *Object-Oriented Analysis and Design With Applications.* Redwood City: Benjamin/Cummings, 1994^2.

[CEW95] Carstensen, Martin; Ebert, Jürgen; Winter, Andreas; *Entity-Relationship-Diagramme und Graphklassen.*
Koblenz: Universität Koblenz-Landau, Fachberichte Informatik, 1995.

[EF94] Ebert, Jürgen; Franzke, Angelika; *A Declarative Approach to Graph Based Modeling.* In: Mayr, E.; Schmidt, G.; Tinhofer, G. [Eds.]; Graph-theoretic Concepts in Computer Science, Lecture Notes in Computer Science, LNCS 903, p. 38-50. Berlin: Springer, 1995.

[EWD+96] Ebert, Jürgen; Winter, Andreas; Dahm, Peter; Franzke, Angelika; Süttenbach, Roger; *Graph Based Modeling and Implementation with EER/GRAL.* In: B. Thalheim [Ed.]; 15th International Conference on Conceptual Modeling (ER'96), Lecture Notes In Computer Science, Proceedings, LNCS 1157. Berlin: Springer, 1996.

[F96] Franzke, Angelika; *GRAL: A Reference Manual.* Koblenz: Universität Koblenz-Landau, Fachbericht Informatik 11/96, 1996.

[S92] Spivey, J.M.; *The $\mathcal{Z}$ Notation.* A Reference Manual. Prentice Hall: Hemel Hempstead, 1992^2.

A The Overall Metamodel

The EER description in figure 5 and the following collection of GRAL predicates summarize the logical, the dynamic and physical view, and express the metamodel of the Booch method. Equal concepts which are used in several views are merged into one concept and are assigned to that view in which they are appeared at first.

The *logical view* which comprises

- class diagrams and
- specifications

is expressed by all the specialized concepts of the abstract concept CDE.

The *dynamic view* which comprises

- state transistion diagrams,
- object diagrams, and
- interaction diagrams

is expressed by all the specialized concepts of the abstract concepts STDE and ODE.

The *physical view* which comprises

- module diagrams, and
- process diagrams

is expressed by all the specialized concepts of the abstract concepts MDE and PDE.

The *integration* between these views is expressed by merged entity types and relationships between these views, for example the mapsTo relationship between an Event of the dynamic view and the Operation of the logical view.

The *metamodel* of the Booch method consists of

- 33 entity types,
- 64 relationship types,
- 56 attributes,
- 12 generalizations,
- 7 aggregations, and
- 42 A Link is connected to two Objects by GRAL predicates

in its entirety.

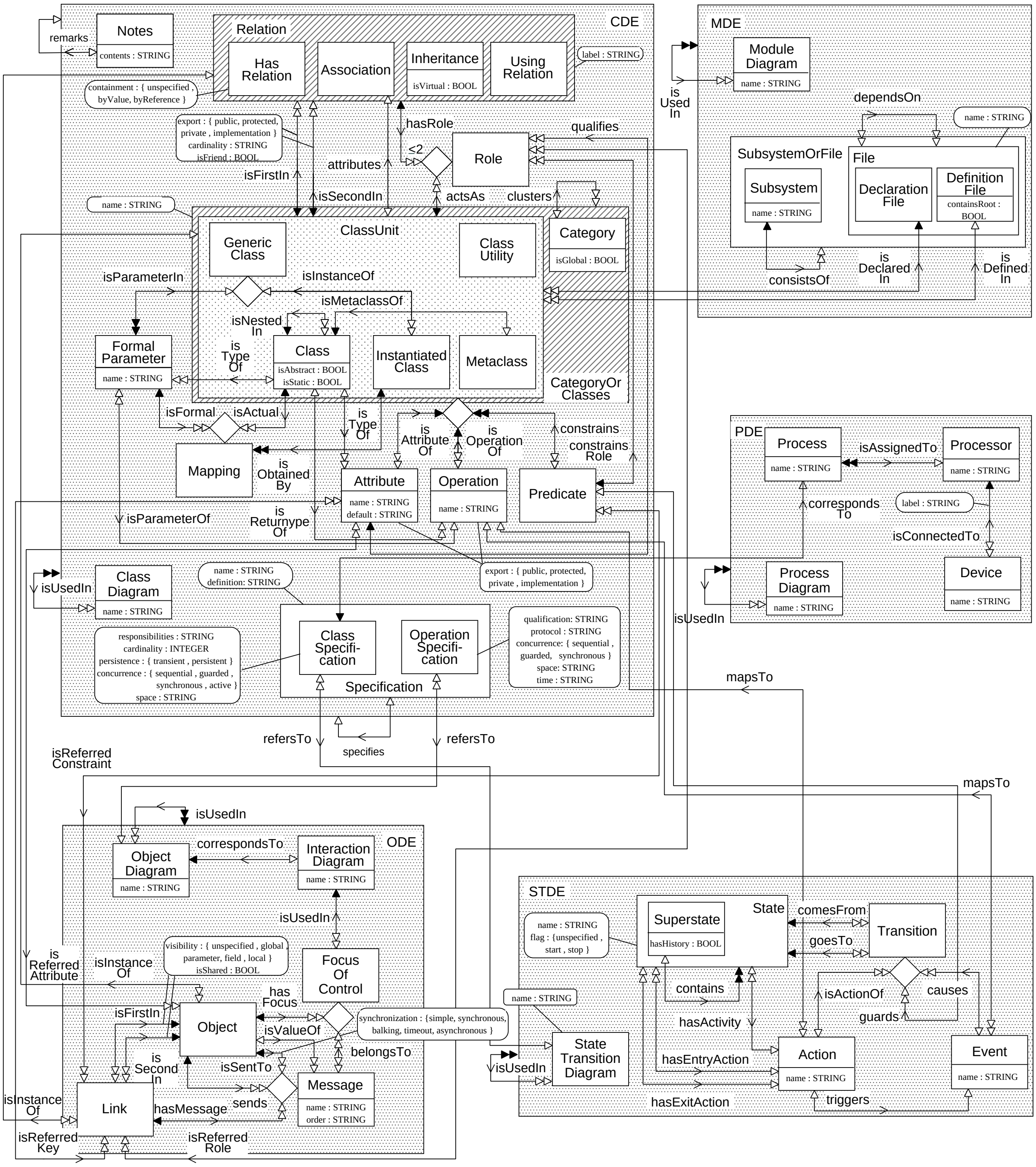


Figure 5: Metamodel of the Booch Method

forall G **in** *Booch* **assert**

CD1 : $\forall\, cd_1, cd_2 : V_{ClassDiagram} \mid cd_1 \neq cd_2 \bullet cd_1.name \neq cd_2.name \;;$

CD2 : $\forall\, cg_1, cg_2 : V_{Category} \mid cg_1 \neq cg_2 \bullet cg_1.name \neq cg_2.name \;;$

CD3 : $\forall\, cg : V_{Category};\; c : V_{ClassUnit} \bullet cg.name \neq c.name \;;$

CD4 :
$$\begin{array}{l} \forall\, c_1, c_2 : V_{ClassUnit} \mid c_1 \neq c_2 \\ \quad \bullet\ (c_1.name \neq c_2.name) \vee \\ \quad\ (\exists\, cg : V_{Category} \bullet cg \rightharpoonup_{clusters} c_1 \wedge \neg\, (cg \rightharpoonup_{clusters} c_2)) \vee \\ \quad\ (\exists\, c : V_{Class} \bullet c \leftharpoonup_{isNestedIn} c_1 \wedge \neg\, (c \leftharpoonup_{isNestedIn} c_2)) \;; \end{array}$$

CD5 : $isForest(eGraph(\rightharpoonup_{clusters})) \;;$

CD6 :
$$\begin{array}{l} \forall\, ic : V_{InstantiatedClass};\; gc : V_{GenericClass} \mid ic \rightharpoonup_{isInstanceOf} gc \\ \quad \bullet\ (\forall\, fp : V_{FormalParameter} \mid fp \rightharpoonup_{isParameterIn} gc \\ \qquad \bullet\ fp \rightharpoonup_{isFormal} \leftharpoonup_{isObtainedBy} ic\ ) \;; \end{array}$$

CD7 : $\forall\, cg : V_{Category} \bullet (cg \rightharpoonup_{isFirstIn}\ \cup\ cg \rightharpoonup_{isSecondIn})\ \subseteq V_{UsingRelation} \;;$

CD8 :
$$\begin{array}{l} \forall\, c_1, c_2 : V_{ClassUnit} \mid c_1 \rightharpoonup_{isFirstIn} \leftharpoonup_{isSecondIn} c_2 \\ \quad \bullet\ (degree(\leftharpoonup_{clusters}, c_1) = 0 \wedge degree(\leftharpoonup_{clusters}, c_2) = 0) \vee \\ \quad\ (\exists\, cg : V_{Category} \bullet cg(\rightharpoonup_{clusters})^{+} c_2 \wedge cg.isGlobal = TRUE) \vee \\ \quad\ (\exists\, cg : V_{Category} \bullet cg(\rightharpoonup_{clusters})^{+} c_1 \leftharpoonup_{clusters} (\rightharpoonup_{clusters})^{+} c_2) \vee \\ \quad\ (c_1(\leftharpoonup_{clusters})^{+} \rightharpoonup_{isFirstIn} \bullet_{UsingRelation} \leftharpoonup_{isSecondIn} (\rightharpoonup_{clusters})^{+} c_2) \;; \end{array}$$

CD9 : $\forall\, v : V_{ClassUnit} \bullet \neg\, (c\ (\rightharpoonup_{isFirstIn} \bullet_{Inheritance} \leftharpoonup_{isSecondIn})^{+} c) \;;$

CD10 :
$$\begin{array}{l} \forall\, ic : V_{InstantiatedClass} \\ \quad \bullet\ (\forall\, m : V_{Mapping} \mid ic \rightharpoonup_{isObtainedBy} m \\ \qquad \bullet\ (m \leftharpoonup_{isActual} (\rightharpoonup \bullet_{Association} \leftharpoonup \mid \\ \qquad \rightharpoonup_{isSecondIn} \bullet_{UsingRelation} \leftharpoonup_{isFirstIn})\ ic)) \;; \end{array}$$

CD11 :
$$\begin{array}{l} \forall\, ro : V_{Role};\; a : V_{Attribute};\; c : V_{ClassUnit} \mid a \rightharpoonup_{qualifies} ro \leftharpoonup_{actsAs} c \\ \quad \bullet\ (\exists\, c_1 : V_{ClassUnit};\; r : V_{Relation} \mid r \rightharpoonup_{hasRole} ro \\ \qquad \bullet\ (c \rightharpoonup_{isFirstIn} r \leftharpoonup_{isSecondIn} c_1 \leftharpoonup_{isAttributeOf} a) \vee \\ \qquad (c \rightharpoonup_{isSecondIn} r \leftharpoonup_{isFirstIn} c_1 \leftharpoonup_{isAttributeOf} a)) \;; \end{array}$$

CD12 : $$\forall r_1, r_2 : V_{Relation} \mid r_1.name = r_2.name \\ \bullet\ r_1 = r_2 \lor (\exists\, c : V_{ClassUnit} \bullet (c \rightharpoonup r_1) \land \neg\, (c \rightharpoonup r_2))\ ;$$

CD13 : $$\forall c : V_{Class} \mid degree(\leftharpoonup_{isNestedIn}, c) > 0 \\ \bullet\ degree(\rightharpoonup_{isFirstIn}, c) = 0 \land degree(\rightharpoonup_{isSecondIn}, c) = 0\ ;$$

CD14 : $$isForest(eGraph(\leftharpoonup_{isNestedIn}))\ ;$$

CD15 : $$\forall cs : V_{ClassSpecification} \bullet (cs \rightharpoonup_{specifies}) \subseteq V_{ClassUnit}\ ;$$

CD16 : $$\forall os : V_{OperationSpecification} \bullet (os \rightharpoonup_{specifies}) \subseteq V_{Operation}\ ;$$

STD1 : $$\forall std_1, std_2 : V_{StateTransitionDiagram} \mid std_1 \neq std_2 \\ \bullet\ std_1.name \neq std_2.name\ ;$$

STD2 : $$\forall s_1, s_2 : V_{State} \mid s_1 \neq s_2 \bullet s_1.name \neq s_2.name\ ;$$

STD3 : $$\forall e_1, e_2 : V_{Event} \mid e_1.name = e_2.name \\ \bullet\ e_1 = e_2 \lor \neg\, (e_1 \rightharpoonup_{causes} \rightharpoonup_{comesFrom} \leftharpoonup_{comesFrom} \leftharpoonup_{causes}\ e_2)\ ;$$

STD4 : $$\forall s : V_{State} \bullet s.flag = stop \land degree(\leftharpoonup_{comesFrom}, s) = 0\ ;$$

STD5 : $$\forall su : V_{Superstate} \\ \bullet\ \{s : V_{State} \mid su \rightharpoonup_{contains} s \land s.flag = start\} \leq 1\ ;$$

STD6 : $$\forall su : V_{Superstate} \mid degree(\leftharpoonup_{goesTo}, su) > 0 \\ (\exists_1\, s : V_{State} \bullet su \rightharpoonup_{contains} s \land s.flag = start\ )\ ;$$

STD7 : $$isForest(eGraph(\rightharpoonup_{contains}))\ ;$$

STD8 : $$\forall std : V_{StateTransitionDiagram} \\ \bullet\ (\exists_1\, s : V_{State} \bullet s \rightharpoonup_{isUsedIn} std \land \mathrm{degree}(\leftharpoonup_{contains}, s) = 0 \land \\ s.flag = start)\ ;$$

STD9 : $$\forall e : V_{Event};\ o : V_{Operation} \mid e \rightharpoonup_{mapsTo} o \\ \bullet\ e \rightharpoonup_{isUsedIn} \leftharpoonup_{refersTo} \rightharpoonup_{specifies} \leftharpoonup_{isOperationOf}\ o\ ;$$

STD10 : $$\forall a : V_{Action} \bullet degree(\rightharpoonup_{mapsTo}, a) + degree(\rightharpoonup_{triggers}, a) = 1\ ;$$

OD1 : $\forall\, od_1, od_2 : V_{ObjectDiagram} \mid od_1 \neq od_2 \bullet od_1.name \neq od_2.name\ ;$

OD2 : $\forall\, o : V_{Object};\ a : V_{Attribute} \mid o \leftharpoonup_{isReferredAttribute}\ a$
$\bullet\ o \rightharpoonup_{isInstanceOf} \leftharpoonup_{isAttributeOf}\ a\ ;$

OD3 : $\forall\, l : V_{Link};\ ro : V_{Role} \mid l \leftharpoonup_{isReferredRole}\ ro$
$\bullet\ l \rightharpoonup_{isInstanceOf} \rightharpoonup_{hasRole}\ ro\ ;$

OD4 : $\forall\, l : V_{Link};\ a : V_{Attribute} \mid l \leftharpoonup_{isReferredKey}\ a$
$\bullet\ l \rightharpoonup_{isInstanceOf} \rightharpoonup_{hasRole} \leftharpoonup_{qualifies}\ a\ ;$

OD5 : $\forall\, l : V_{Link};\ p : V_{Predicate} \mid l \leftharpoonup_{isReferredConstraint}\ p$
$\bullet\ l \rightharpoonup_{isInstanceOf} \rightharpoonup_{hasRole} \leftharpoonup_{constrainsRole}\ p\ ;$

OD6 : $\forall\, l : V_{Link};\ o_1, o_2 : V_{Object} \mid l \rightharpoonup_{isFirstIn}\ o_1 \wedge l \rightharpoonup_{isSecondIn}\ o_2$
$\bullet\ (\exists\, r : V_{Relation} \mid l \rightharpoonup_{isInstanceOf}\ r$
$\bullet\ (o_1 \rightharpoonup_{isInstanceOf} \rightharpoonup_{isFirstIn}\ r \leftharpoonup_{isSecondIn} \leftharpoonup_{isInstanceOf}\ o_2) \vee$
$(o_1 \rightharpoonup_{isInstanceOf} \rightharpoonup_{isSecondIn}\ r \leftharpoonup_{isFirstIn} \leftharpoonup_{isInstanceOf}\ o_2))\ ;$

OD7 : $\forall\, id : V_{InteractionDiagram};\ foc : V_{FocusOfControl} \mid id \leftharpoonup_{isUsedIn} foc$
$\bullet\ (\forall\, o : V_{Object} \mid o \rightharpoonup_{hasFocus} foc$
$\bullet\ o \rightharpoonup_{isUsedIn} \leftharpoonup_{correspondsTo}\ id) \wedge$
$(\forall\, m : V_{Message} \mid m \rightharpoonup_{belongsTo} foc$
$\bullet\ m \rightharpoonup_{isUsedIn} \leftharpoonup_{correspondsTo}\ id)\ ;$

MD1 : $\forall\, md_1, md_2 : V_{ModuleDiagram} \mid md_1 \neq md_2$
$\bullet\ md_1.name \neq md_2.name\ ;$

MD2 : $\forall\, sb_1, sb_2 : V_{Subsystem} \mid sb_1 \neq sb_2 \bullet sb_1.name \neq sb_2.name\ ;$

MD3 : $\forall f_1, f_2 : V_{File} \mid f_1 \neq f_2$
$\bullet\ f_1.name \neq f_2.name \vee$
$(type(f_1) = DeclarationFile \wedge type(f_2) = DefinitionFile) \vee$
$(\exists\, sb : V_{Subsystem} \bullet (sb \rightharpoonup_{consistsOf}\ f_1) \wedge \neg\, (sb \rightharpoonup_{consistsOf}\ f_2)\ ;$

MD4 : $\forall\, sb : V_{Subsystem}$
$\bullet\ \{f : V_{DefinitionFile} \mid$
$f.containsRoot = TRUE \wedge sb \rightharpoonup_{consistsOf}\ f\} \leq 1\ ;$

MD5 : $\forall\, md : V_{ModuleDiagram}$
$\quad\bullet (\exists_1 f : V_{DefinitionFile} \mid$
$\qquad f.containsRoot = TRUE \wedge degree(\leftharpoondown_{consistsOf}, f) = 0)\ ;$

MD6 : $\forall f : V_{File} \bullet \neg\, (f (\rightharpoonup_{dependsOn})^{+} f)\ ;$

PD1 : $\forall\, pd_1, pd_2 : V_{ProcessDiagram} \mid pd_1 \neq pd_2$
$\quad\bullet\, pd_1.name \neq pd_2.name\ ;$

PD2 : $\forall\, cs : V_{ClassSpecification} \mid cs.concurrency = active$
$\quad\bullet \exists\, p : V_{Process} \bullet c \leftharpoondown_{correspondsTo}\ cs\ ;$

PD3 : $\forall\, p : V_{Process}$
$\quad\bullet (\exists_1\, cs : V_{ClassSpecification}$
$\qquad\bullet\, p \rightharpoonup_{correspondsTo}\ cs \wedge cs.concurrence = active)\ ;$

Available Research Reports (since 1995):

1997

5/97 *Roger Süttenbach, Jürgen Ebert.* A Booch Metamodel.

4/97 *Jürgen Dix, Luis Pereira, Teodor Przymusinski.* Prolegomena to Logic Programming for Non-Monotonic Reasoning.

3/97 *Angelika Franzke.* GRAL 2.0: A Reference Manual.

2/97 *Ulrich Furbach.* A View to Automated Reasoning in Artificial Intelligence.

1/97 *Chandrabose Aravindan, Jürgen Dix, Ilkka Niemelä .* The DisLoP-Project.

1996

28/96 *Wolfgang Albrecht.* Echtzeitplanung für Alters- oder Reaktionszeitanforderungen.

27/96 *Kurt Lautenbach.* Action Logical Correctness Proving.

26/96 *Frieder Stolzenburg, Stephan Höhne, Ulrich Koch, Martin Volk.* Constraint Logic Programming for Computational Linguistics.

25/96 *Kurt Lautenbach, Hanno Ridder.* Die Lineare Algebra der Verklemmungsvermeidung — Ein Petri-Netz-Ansatz.

24/96 *Peter Baumgartner, Ulrich Furbach.* Refinements for Restart Model Elimination.

23/96 *Peter Baumgartner, Peter Fröhlich, Ulrich Furbach, Wolfgang Nejdl.* Tableaux for Diagnosis Applications.

22/96 *Jürgen Ebert, Roger Süttenbach, Ingar Uhe.* Meta-CASE in Practice: a Case for KOGGE.

21/96 *Harro Wimmel, Lutz Priese.* Algebraic Characterization of Petri Net Pomset Semantics.

20/96 *Wenjin Lu.* Minimal Model Generation Based on E-Hyper Tableaux.

19/96 *Frieder Stolzenburg.* A Flexible System for Constraint Disjunctive Logic Programming.

18/96 *Ilkka Niemelä (Ed.).* Proceedings of the ECAI'96 Workshop on Integrating Nonmonotonicity into Automated Reasoning Systems.

17/96 *Jürgen Dix, Luis Moniz Pereira, Teodor Przymusinski.* Non-monotonic Extensions of Logic Programming: Theory, Implementation and Applications (Proceedings of the JICSLP '96 Postconference Workshop W1).

16/96 *Chandrabose Aravindan.* DisLoP: A Disjunctive Logic Programming System Based on PROTEIN Theorem Prover.

15/96 *Jürgen Dix, Gerhard Brewka.* Knowledge Representation with Logic Programs.

14/96 *Harro Wimmel, Lutz Priese.* An Application of Compositional Petri Net Semantics.

13/96 *Peter Baumgartner, Ulrich Furbach.* Calculi for Disjunctive Logic Programming.

12/96 *Klaus Zitzmann.* Physically Based Volume Rendering of Gaseous Objects.

11/96 *J. Ebert, A. Winter, P. Dahm, A. Franzke, R. Süttenbach.* Graph Based Modeling and Implementation with EER/GRAL.

10/96 *Angelika Franzke.* Querying Graph Structures with G^2QL.

9/96 *Chandrabose Aravindan.* An abductive framework for negation in disjunctive logic programming.

8/96 *Peter Baumgartner, Ulrich Furbach, Ilkka Niemelä .* Hyper Tableaux.

7/96 *Ilkka Niemelä, Patrik Simons.* Efficient Implementation of the Well-founded and Stable Model Semantics.

6/96 *Ilkka Niemelä .* Implementing Circumscription Using a Tableau Method.

5/96 *Ilkka Niemelä .* A Tableau Calculus for Minimal Model Reasoning.

4/96 *Stefan Brass, Jürgen Dix, Teodor. C. Przymusinski.* Characterizations and Implementation of Static Semantics of Disjunctive Programs.

3/96 *Jürgen Ebert, Manfred Kamp, Andreas Winter.* Generic Support for Understanding Heterogeneous Software.

2/96 *Stefan Brass, Jürgen Dix, Ilkka Niemelä, Teodor. C. Przymusinski.* A Comparison of STATIC Semantics with D-WFS.

1/96 *J. Ebert (Hrsg.).* Alternative Konzepte für Sprachen und Rechner, Bad Honnef 1995.

1995

21/95 *J. Dix and U. Furbach.* Logisches Programmieren mit Negation und Disjunktion.

20/95 *L. Priese, H. Wimmel.* On Some Compositional Petri Net Semantics.

19/95 *J. Ebert, G. Engels.* Specification of Object Life Cycle Definitions.

18/95 *J. Dix, D. Gottlob, V. Marek.* Reducing Disjunctive to Non-Disjunctive Semantics by Shift-Operations.

17/95 *P. Baumgartner, J. Dix, U. Furbach, D. Schäfer, F. Stolzenburg.* Deduktion und Logisches Programmieren.

16/95 *Doris Nolte, Lutz Priese.* Abstract Fairness and Semantics.

15/95 *Volker Rehrmann (Hrsg.).* 1. Workshop Farbbildverarbeitung.

14/95 *Frieder Stolzenburg, Bernd Thomas.* Analysing Rule Sets for the Calculation of Banking Fees by a Theorem Prover with Constraints.

13/95 *Frieder Stolzenburg.* Membership-Constraints and Complexity in Logic Programming with Sets.

12/95 *Stefan Brass, Jürgen Dix.* D-WFS: A Confluent Calculus and an Equivalent Characterization..

11/95 *Thomas Marx.* NetCASE — A Petri Net based Method for Database Application Design and Generation.

10/95 *Kurt Lautenbach, Hanno Ridder.* A Completion of the S-invariance Technique by means of Fixed Point Algorithms.

9/95 *Christian Fahrner, Thomas Marx, Stephan Philippi.* Integration of Integrity Constraints into Object-Oriented Database Schema according to ODMG-93.

8/95 *Christoph Steigner, Andreas Weihrauch.* Modelling Timeouts in Protocol Design..

7/95 *Jürgen Ebert, Gottfried Vossen.* I-Serializability: Generalized Correctness for Transaction-Based Environments.

6/95 *P. Baumgartner, S. Brüning.* A Disjunctive Positive Refinement of Model Elimination and its Application to Subsumption Deletion.

5/95 *P. Baumgartner, J. Schumann.* Implementing Restart Model Elimination and Theory Model Elimination on top of SETHEO.

4/95 *Lutz Priese, Jens Klieber, Raimund Lakmann, Volker Rehrmann, Rainer Schian.* Echtzeit-Verkehrszeichenerkennung mit dem Color Structure Code — Ein Projektbericht.

3/95 *Lutz Priese.* A Class of Fully Abstract Semantics for Petri-Nets.

2/95 *P. Baumgartner, R. Hähnle, J. Posegga (Hrsg.).* 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods — Poster Session and Short Papers.

1/95 *P. Baumgartner, U. Furbach, F. Stolzenburg.* Model Elimination, Logic Programming and Computing Answers.