An OMT Metamodel

Jürgen Ebert, Roger Süttenbach

13/97

An OMT Metamodel

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

Abstract

This paper provides an integrated and formalized description of the abstract syntax of the notations of the Object Modeling Technique (OMT) by using the EER/GRAL approach of modeling.

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

1 Introduction

1.1 Metamodel

This paper provides a formalized and integrated description of the abstract syntax of the notations of the *Object Modeling Technique* (OMT) described in [RBP+91]. Strictly speaking, we present a model for the three parts of OMT which are

- the object model,
- the dynamic model, and
- the functional model.

The summarized model is called an *OMT metamodel*. This term refers to the fact that instances of this model are models themself. Its formal basis is the EER/GRAL approach of modeling which we briefly describe in Section 1.2.

The OMT metamodel defines the set of *syntax graphs* of OMT diagrams which are used for modeling real system. A syntax graph of an OMT diagram does not describe the concrete syntax of this model but its abstract structure. For every model, expressed by OMT diagrams, there is one abstract syntax graph. This syntax graph can be checked whether it is correct with respect to the OMT metamodel or not. Thus, an OMT diagram is a correct diagram if its corresponding syntax graph is an instance of the OMT metamodel.

More information about the reasons for formalizing an object-oriented method like OMT and about our modeling principles can be found in [SE97] which is a description of the Booch method [B94] similar to this one.

1.2 EER/GRAL Approach

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 Graphs as described in [EWD+96] and [EF94]. Graphs are used to achieve a formal modeling which has been applied to various fields in software engineering. Graph classes – 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 of OMT.

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

1.2.1 EER

EER descriptions [CEW94] may contain five different elements – *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* shown in Figure 1:

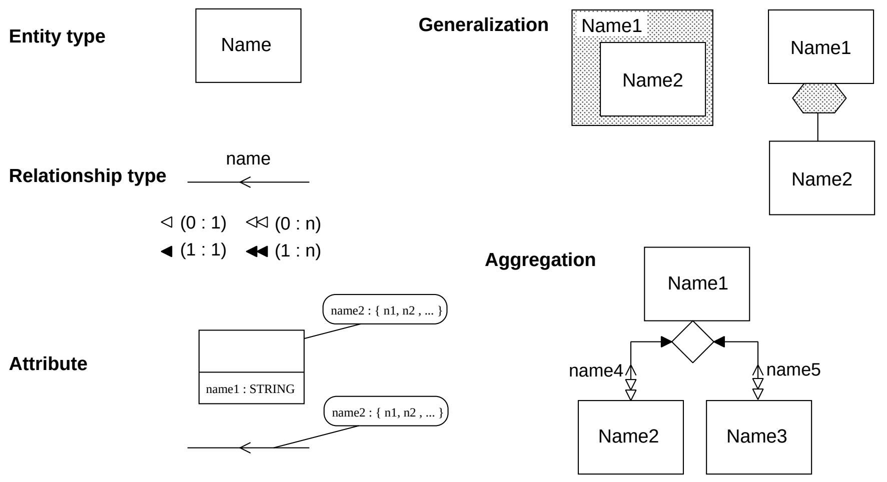


Figure 1: EER notation

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

Entity types are used to describe concepts of the modelled method. That means they represent information which cannot be derived or built from other concepts. Relationship types express which concepts can be connected to each other. The properties of concepts and connections are described by attributes. They describe certain aspects and are not concepts themselves. Note that in our notation relationships may have attributes too. 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. Aggregation allows ternary and higher relationships to be factored into binary ones. This makes the model simpler to understand due to the separation of concerns.

1.2.2 GRAL

The $\mathcal{Z}$-like [S92] *assertion language GRAL* is used to denote such integrity conditions which cannot be expressed by the EER description, for example constraints on the values of the attributes of vertices and edges.

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 [F97].

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

❈

Restrictions on certain vertices, edges or attributes can be expressed easily as the following example[1] shows:

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

$$\mathsf{DM1}: \quad \forall\, sd_1, sd_2 : V_{StateDiagram} \mid sd_1 \neq sd_2 \bullet sd_1.name \neq sd_2.name \;;$$

This means each vertex of type StateDiagram 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:

Control flows are only allowed between processes.

$$\mathsf{FM4}: \quad \forall\, c : V_{ControlFlow} \bullet c \rightharpoonup_{startsAt} \cup\, c \rightharpoonup_{endsAt} \subseteq V_{Process} \;;$$

This means for all vertices of type ControlFlow the set of vertices which is reachable via a direct outgoing edge of type startsAt or of type endsAt must

[1] The abbreviation in front of the GRAL predicates refers to the corresponding place in this paper. For example, DM1 refers to the first constraint of the dynamic model.

be a subset of the set of all vertices of type Process.

❋

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 an inheritance hierarchy.

$$\mathsf{OM5:} \quad \forall\, c : V_{Class} \;\bullet\; \neg\, (c\ (\rightarrow_{isSuperclassIn} \leftarrow_{isSubclassIn})^{+}\ c)\ ;$$

The GRAL predicate above requires that no vertex of type Class is the starting point of a path via outgoing edges of type isSuperclassIn and incoming edges of type isSubclassIn back to itself.

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 edges regarding a certain vertex:

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

$$\begin{aligned}\mathsf{DM7:} \quad & \forall\, su : V_{Superstate} \mid degree(\leftharpoonup_{goesTo}, su) > 0 \\ & \quad \bullet\ (\exists_1\, s : V_{State} \bullet su \rightharpoonup_{contains} s \wedge s.flag = initial\ )\ ;\end{aligned}$$

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 'initial' in its attribute flag and is connected to its superstate by an outgoing edge of type contains.

Structure of This Paper

In **Chapter 2 Object Model**, **3 Dynamic Model** and **4 Functional Model** we incrementally describe each part of OMT. We are guided in this order by the quick reference presented on the first and last two pages of the book [RBP+91]. Each notation 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 meaning of the notations and their graphical representation above the figure, and explanations with respect to the EER description below it. In order to distinguish previously mentioned concepts in the EER description 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.

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

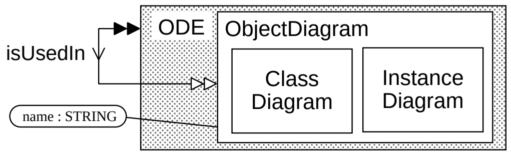
2 Object Model

In OMT the *object model* of a system describes the structure of objects in the system – “their identity, their relationships to other objects, their attributes and their operations” (p. 17)[2]. The object model is represented by a set of *object diagrams* which provide a formal graphical notation.

2.1 Object Diagram

An *object diagram* is a named sheet of paper showing one or more symbols which represent all or a part of the object model of a system. There are two kinds of object diagrams: *class diagrams* and *instance diagrams.* A class diagram describes the general case in modeling a system whereas an instance diagram is used to illustrate a special situation by example.

An object diagram has no explicit notation in OMT. One may take the sheet of paper as the graphical representation.



An ObjectDiagram comprises all elements which are assigned to it by the isUsedIn relationship. Here, the abstract concept object diagram element, ODE for short, is used as a placeholder for these elements which have to be specialized concepts of the ODE. For graphical simplicity, this is done in Figure 2 (p. 19) for the first time. Object diagrams can be explicitly used as ClassDiagrams or InstanceDiagrams.

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

$$\mathsf{OM1}: \quad \forall\, od_1, od_2 : V_{ObjectDiagram} \mid od_1 \neq od_2 \bullet od_1.name \neq od_2.name \;;$$

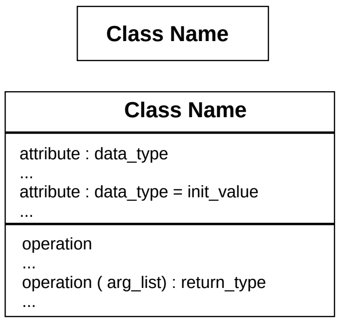
❈

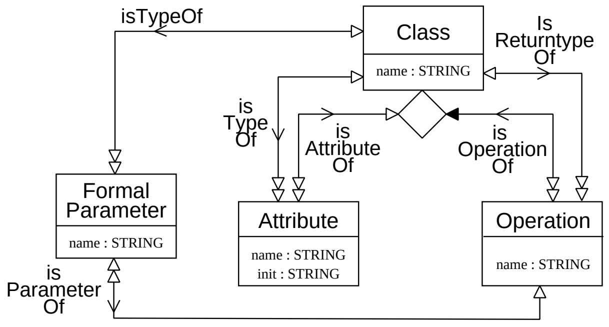
Class:

A *class* “describes a group of objects with similar properties (attributes), common behavior (operations), common relationships to other objects, and common semantics” (p. 22).

The OMT notation for a class is a box which may have as many as three parts. The first part contains the class name, the second part the attributes and the third part contains the operations.

[2] The page numbers refer to the book [RBP+91].


Class Name
Class Name
attribute : data_type
...
attribute : data_type = init_value
...
operation
...
operation (arg_list) : return_type
...



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 an init value. The description of a signature of an operation is similar to the Pascal syntax. Each operation possibly has a return class and has zero or more FormalParameters having a class associated as their type.

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

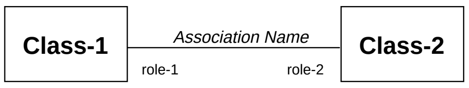
$$\text{OM2}: \quad \forall\, c_1, c_2 : V_{Class} \mid c_1 \neq c_2 \bullet c_1.name \neq c_2.name \;;$$

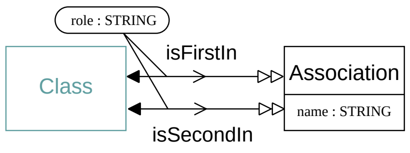
❋

Association:

An *association* describes a group of links – physical or conceptual connections between objects – with common structure and common semantics. A role describes the special context in which the class acts in this association. A role name uniquely identifies one end of an association.

The OMT notation for an association is a line between classes labeled by its name. Although associations are inherently bidirectional, their name „usually reads in a particular direction“ (p. 27). This direction is derived from the positions of the connected classes (left to right or top to bottom). Role names are positioned near the ends of the line.





An Association has one start and one end class which is expressed by the isFirstIn and the isSecondIn relationships. By convention the positions of the classes and therefore the read direction of the association corresponds to these relationships. Roles are modelled by the role attribute of these relationships. Note that here we model the binary association which is connected to two classes only.

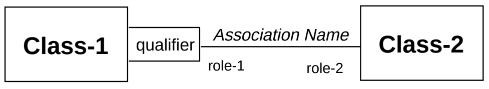
Constraint The two role names of an association have to be different.

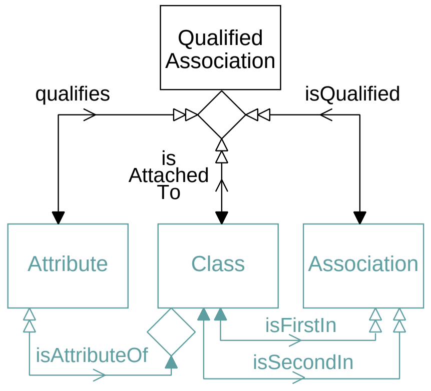
$$\mathsf{OM3:}\quad \forall f : E_{isFirstIn};\ f_1 : E_{isSecondIn} \mid \omega(f) = \omega(f_1) \bullet f.name \neq f_1.name\ ;$$

Qualified Association:

A *qualified association* is an association with an attribute whose values uniquely identify objects. The attribute, called qualifier, is used to navigate in the set of objects denoted by the association.

A qualified association is represented by a line and a small box at one of its end. The box, in which the qualifier is drawn, is positioned near the class which uses the qualifier.





A QualifiedAssociation always consists of three parts: an Association which isQualified, an Attribute which qualifies the association and a Class which isAttachedTo the association.

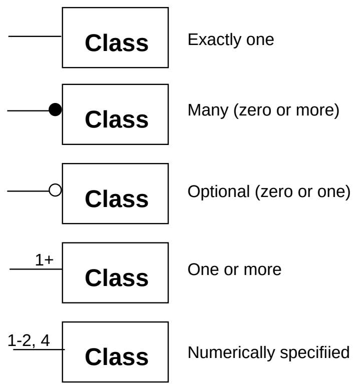
Constraint The qualifier must be an attribute of the other class connected to the association.

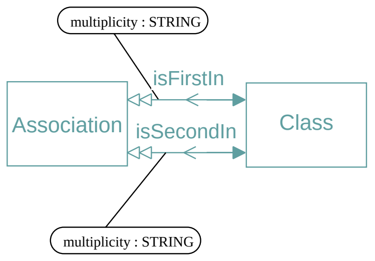
$$\mathsf{OM4:}\quad \forall q : V_{QualifiedAssociation};\ a : V_{Attribute};\ c : V_{Class} \mid a \rightharpoonup_{qualifies} q \leftharpoonup_{isAttachedTo} c \bullet (\exists c_1 : V_{Class};\ r : V_{Association} \mid r \rightharpoonup_{isQualified} q \bullet (c \rightharpoonup_{isFirstIn} r \leftharpoonup_{isSecondIn} c_1 \leftharpoonup_{isAttributeOf} a) \vee (c \rightharpoonup_{isSecondIn} r \leftharpoonup_{isFirstIn} c_1 \leftharpoonup_{isAttributeOf} a))\ ;$$

Multiplicity of Association:

The *multiplicity* of an association describes “how many instances of one class may relate to a single instance of an associated class” (p. 30). Multiplicity is often called cardinality.

Multiplicity is represented by special symbols at the ends of association lines, for example hollow or solid bullets, or textual annotations, for example 1+ or 1-2,4.





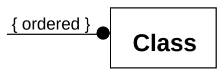
Multiplicity is modelled by the multiplicity attribute of the isFirstIn and isSecondIn relationships. These relationships express the "ends" of associations.

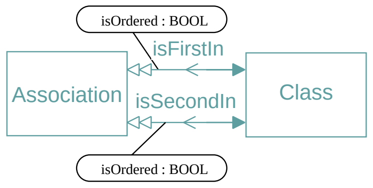
❋

Ordering:

An association may be *ordered*, i.e. the objects described by this association have an explicit order.

The ordering is represented by writing '{ordered}' on the association line next to the class with repect to which it is ordered.





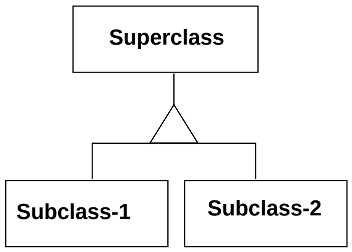
An end of an association isOrdered or is not. The isFirstIn respectively the isSecondIn relationship expresses the "end" of the association which can be ordered.

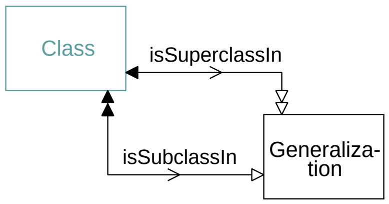
❋

Generalization (Inheritance):

Generalization is a relation between a class and one or more refined versions of it. These specialized classes are called the subclasses whereas the generalized class is called the superclass.

The OMT notation for generalization is a triangle connecting the related classes. The superclass is connected by a line to its top and the subclasses are connected by lines to a horizontal bar attached to its base.





The superclass in a Generalization is given by the isSuperclassIn relationship. Analogously, the subclasses are expressed by the isSubclassIn relationship. Note that in a generalization there exists only one superclass and at least one subclass.

Constraint Cycles are not allowed in an inheritance hierarchy.

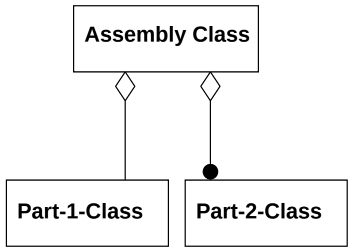
OM5 : $\forall\, c : V_{Class} \bullet \neg\, (c\ (\rightarrow_{isSuperclassIn} \leftarrow_{isSubclassIn})^{+}\ c)$;

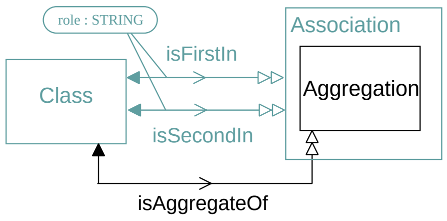
❈

Aggregation:

Aggregation is a relation between a class and one or more classes which are parts of it. These part classes are called component classes whereas the assembled class is called the aggregate class. In OMT the aggregation "is a tightly coupled form of association with some extra semantics" (p. 37) and "not an independent concept" (p. 58).

The OMT notation for aggregation is a small diamond directly connected to the aggregate and by a line to the component classes. At the end of the line the multiplicity is annotated.





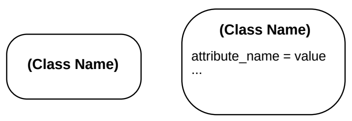
Since Aggregation is a special form of Association we use generalization to model this aspect. The aggregate class in an aggregation is indicated by the isAggregateOf relationship. Note that an aggregation automatically inherits the isFirstIn and isSecondIn relationships because it is a specialized concept of association.

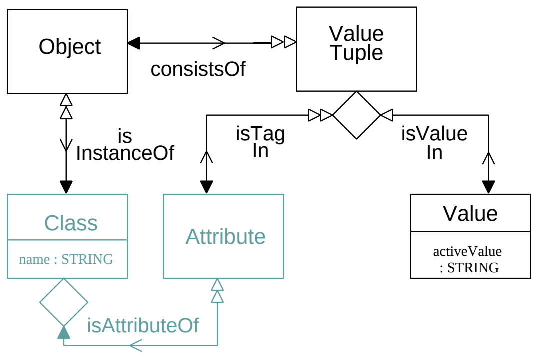
❈

Object Instances:

An *object instance* is “a concept, abstraction, or thing with crisp boundaries and meanings for the problem at hand” (p. 21). Additionally, an object is an instance of a class.

An object is represented by a rounded box. The class name is written in round parentheses at the top of the box. Within the box attribute-value pairs may be listed.





An object instance is modelled by the concept Object which is an InstanceOf a Class. The attribute-value pairs are expressed by ValueTuples which always comprise an Attribute and a Value.

Constraint Each attribute within an object must be an attribute of the object’s class.

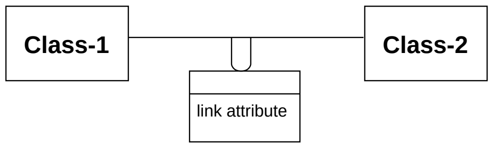
$$\mathsf{OM6}: \quad \forall\, o : V_{Object};\ t : V_{ValueTuple} \mid o \rightharpoonup_{consistsOf} t \bullet t \leftharpoonup_{isTagIn} \rightharpoonup_{isAttributeOf} \leftharpoonup_{isInstanceOf} o\ ;$$

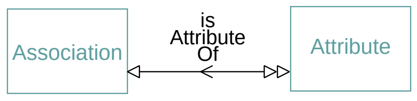
❋

Link Attribute:

A *link attribute* is a property of the links in an association, similarly to the attributes of the objects in a class.

The OMT notation for a link attribute is a box attached to the association by a loop in which the attributes are listed in the second part of the box.





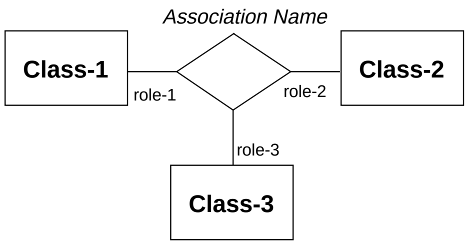
A link attribute is modelled by an Attribute which is assigned to an association by the isAttributeOf relationship.

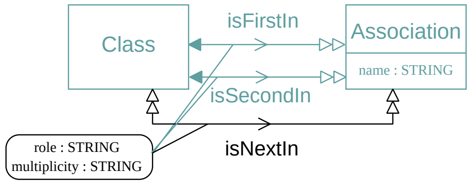
❋

Ternary Association:

A *ternary association* is an association among three classes. Analogously, an n-ary association is an association among n classes.

The OMT notation for ternary and n-ary associations is a diamond with lines connecting to related classes.





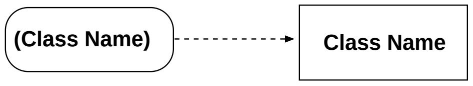
The ternary and the n-ary association is expressed by the isNextIn relationship which is used to connect the additional classes. In this case the isFirstIn and isSecondIn relationships express no read direction but they guarantee correct modeling because there are always two classes connected to an association.

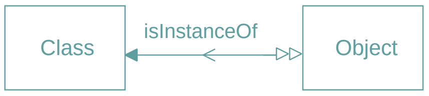
❊

Instance Relationship:

An *instance relationship* “relates a class to its instances” (p. 69) which are objects.

An instance relationship is represented by a dotted arrow connecting the instance to the class.





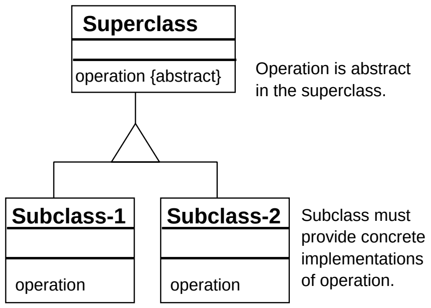
The instantiation relationship between Classes and its Objects is already modelled on page 11 by the isInstanceOf relationship.

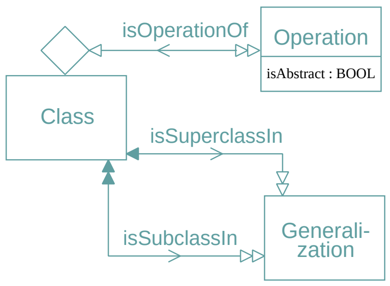
❊

Abstract Operation:

An *abstract operation* “defines the form of an operation for which each concrete subclass must provide its own implementation” (p. 62). Abstract operations are used to make sure that all subclasses have a common interface.

An abstract operation is represented by writing ‘{abstract}’ behind the signature of the operation.


Superclass
operation {abstract}
Operation is abstract in the superclass.
Subclass-1
operation
Subclass-2
operation
Subclass must provide concrete implementations of operation.



An abstract operation is modelled by a boolean attribute which express whether an operation is an Abstract operation or is not.

Constraint Abstract operations are only allowed in superclasses.

$$\text{OM7}: \quad \forall\, p : V_{Operation};\ c : V_{Class} \mid p.isAbstract = TRUE \wedge p \rightharpoonup_{isOperationOf} c \bullet degree(\rightharpoonup_{isSuperclassIn}, c) > 0\ ;$$

Constraint There must exist corresponding concrete operations in the subclasses as implementation for the abstract operation in the superclass.

$$\text{OM8}: \quad \forall\, p : V_{Operation};\ c : V_{Class} \mid p.isAbstract = TRUE \wedge p \rightharpoonup_{isOperationOf} c \bullet (\forall\, c_1 : V_{Class} \mid c\ (\rightharpoonup_{isSuperclassIn} \leftharpoondown_{isSubclassIn})^{+} c_1 \bullet (\exists\, p_1 : V_{Operation} \mid p_1 \rightharpoonup_{isOperationOf} c_1 \bullet p.name = p_1.name \wedge p_1.isAbstract = FALSE))\ ;$$

Constraint Abstract operations are not allowed in a class if a concrete operation already exists in one of its superclasses.

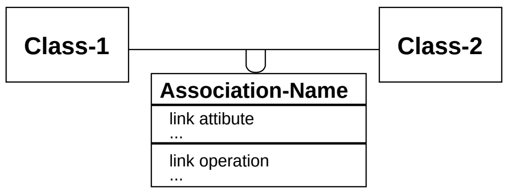
$$\text{OM9}: \quad \forall\, p : V_{Operation} \mid p.isAbstract = TRUE \bullet (\forall\, p_1 : V_{Operation} \mid p \rightharpoonup_{isOperationOf} (\rightharpoonup_{isSuperclassIn} \leftharpoondown_{isSubclassIn})^{+} \leftharpoondown_{isOperationOf} p_1 \bullet p.name \neq p_1.name);$$

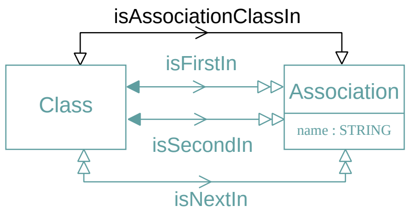
❈

Association as Class:

In OMT an *association* can be modelled as a *class*, i.e. each “link becomes one instance of the class” (p. 33).

The OMT notation for modeling an association as a class is a box, like the box for classes, attached to the association by a loop.





The modeling of an association as a class is expressed by a Class which is a AssociationClassIn an Association.

Constraint The modeling of an association as a class is only allowed in binary associations.

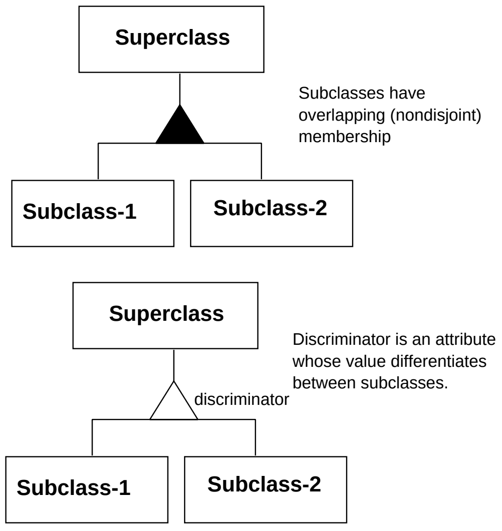
$$\text{OM10}: \quad \forall\, a : V_{Association} \mid degree(\leftharpoondown_{isAssociationClassIn}, a) > 0 \bullet degree(\leftharpoondown_{isNextIn}, a) = 0\ ;$$

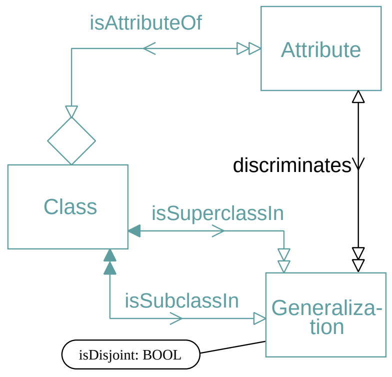
❈

Generalization Properties:

A generalization is a *disjoint* generalization if all objects of the superclass are only instances of *one* of its subclass. If there is an object which belongs to two subclasses at the same time the generalization is called *nondisjoint*. A *discriminator* is an attribute of the superclass whose values can be used to divide it into the subclasses. The discriminator is “simply a name for the basis of generalization” (p. 39).

A nondisjoint generalization is represented by a filled triangle instead of a hollow one which represents the disjoint generalization. A discriminator is represented textually.


Superclass
Subclasses have overlapping (nondisjoint) membership
Subclass-1
Subclass-2
Superclass
Discriminator is an attribute whose value differentiates between subclasses.
discriminator
Subclass-1
Subclass-2



A disjoint generalization is modelled by a boolean attribute which express whether a generalization is a Disjoint generalization or is not. The use of an Attribute as a discriminator is expressed by the discriminates relationship.

Constraint The discriminator in an inheritance hierarchy must be an attribute of the direct superclass.

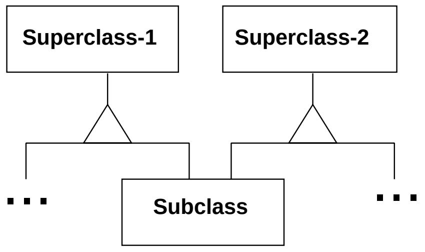
$$\text{OM11}: \quad \forall\, g : V_{Generalization};\ a : V_{Attribute} \mid a \rightharpoonup_{discriminates} g \\ \bullet\ a \rightharpoonup_{isAttribute} \rightharpoonup_{isSuperclassIn} g\ ;$$

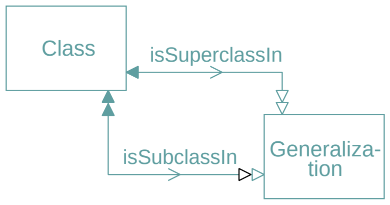
❈

Multiple Inheritance:

Multiple inheritance permits a class to have more than *one* superclasses, or a class is a subclass in serveral generalization hierarchies, respectively.

Multiple inheritance is represented by connecting a class to several triangles which represent different generalizations.





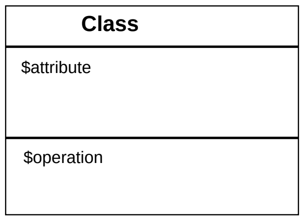
Multiple inheritance is modelled by increasing the cardinality of one end of the isSubclassIn from possibly one to many.

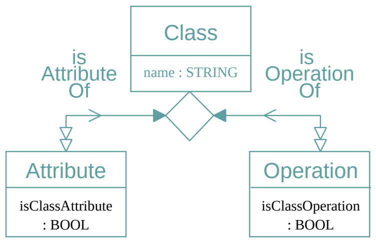
❈

Class Attributes and Class Operations:

A *class attribute* is an attribute which is shared by all objects of this class and is not restricted to a single object. A *class operation* is "an operation on the class itself" (p. 71). Typically, constructor or destructor operations are class operations but also operations which operate on class attributes.

Class attributes and operations are indicated by a leading dollar sign ($).





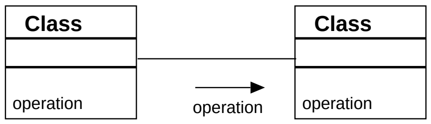
Class attributes and operations are modelled by boolean atributes. An attribute is an ClassAttribute or it is not. An operation is an ClassOperation or it is not.

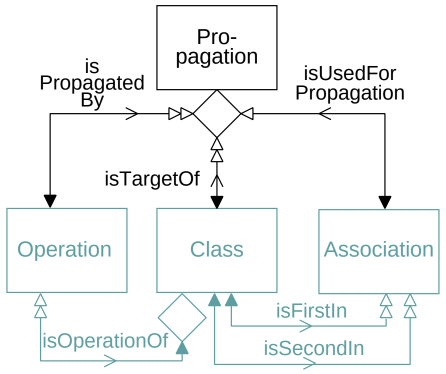
❈

Propagation of Operations:

The *propagation* of an operation is the “automatic application of an operation to a network of objects when the operation is applied to some starting object” (p. 60).

The OMT notation for propagation is a small arrow annotated with a name next to the affected association.





A Propagation always consist of three parts: an Operation which isPropagatedBy this propagation, a Class which is the TargetOf the operation and an Association which isUsedForPropagation.

Constraint The propagated operation must be an operation of the source class which is connected with the association and the target class.

$$\text{OM12}: \quad \forall\, pr : V_{Propagation};\ p : V_{Operation};\ c : V_{Class} \mid p \rightharpoonup_{isPropagatedBy} pr \leftharpoonup_{isTargetOf} c \bullet (\exists\, c_1 : V_{Class};\ a : V_{Association} \mid a \rightharpoonup_{isUsedForPropagation} pr \bullet (c \rightharpoonup_{isFirstIn} a \leftharpoonup_{isSecondIn} c_1 \leftharpoonup_{isOperationOf} p) \vee (c \rightharpoonup_{isSecondIn} a \leftharpoonup_{isFirstIn} c_1 \leftharpoonup_{isOperationOf} p))\ ;$$

Constraint The target class must have an operation with the same name as the propagated operation.

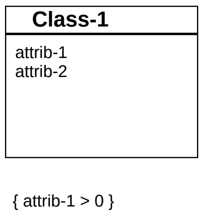
$$\text{OM13}: \quad \forall\, pr : V_{Propagation};\ p : V_{Operation};\ c : V_{Class} \mid p \rightharpoonup_{isPropagatedBy} pr \leftharpoonup_{isTargetOf} c \bullet (\exists\, p_1 : V_{Operation} \bullet p_1 \rightharpoonup_{isOperationOf} c \wedge p.name = p_1.name)\ ;$$

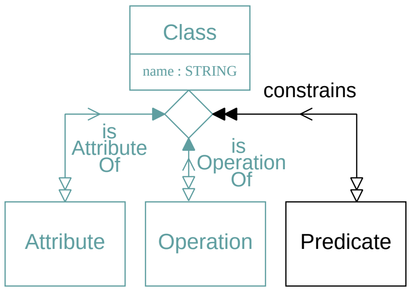
❈

Constraints on Objects:

A *constraint* "is a functional relationship between entities of an object model" (p. 73) such as objects, classes and attributes. Constraints restrict the values that entities can assume.

Constraints are annotated in curved parentheses and are positioned near the symbol for the constrained entity.





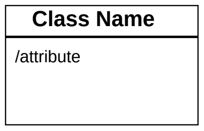
In OMT Predicates are used as constraints. Since predicates are tightly coupled to classes and their elements, mainly attributes and operations, we modelled the Predicate as an component of the concept Class. The concept predicate is not further refined here. In general, they look like "salary =< boss.salary" (p. 74).

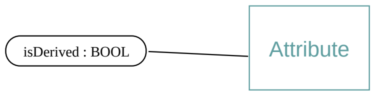
❈

Derived Attribute:

A *derived attribute* is an attribute which can be computed through other attributes.

Derived attributes are indicated by a leading backslash sign (/).





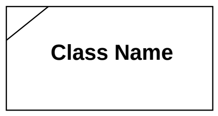
An attribute is a Derived attribute or it is not.

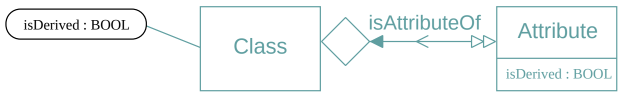
❈

Derived Class:

A *derived class* is a class whose objects are completely determined by objects of another class.

The OMT notation for a derived class is a diagonal line on the upper-left corner of the class box.





A class is a Derived class or it is not.

Constraint A derived class is only allowed to have derived attributes.

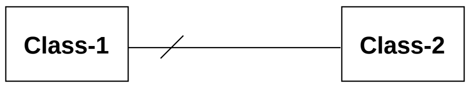
OM14 : $\forall\, c : V_{Class} \mid c.isDerived = TRUE$
$\bullet \forall\, a : V_{Attributes} \mid c \leftharpoonup_{isAttributeOf}\, a \bullet a.isDerived = TRUE$;

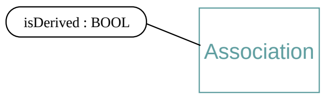
❊

Derived Association:

A *derived association* is an association whose links are completely determined by links of another association.

The OMT notation for a derived association is a diagonal line on the association line.





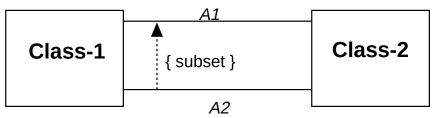
An association is a Derived association or it is not.

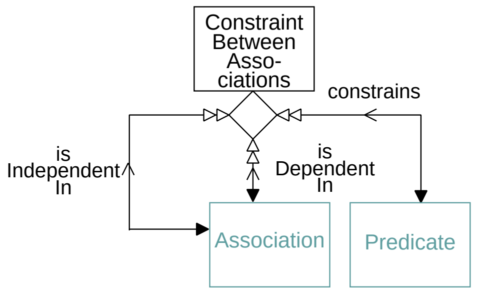
❊

Constraint between Associations:

A *constraint between associations* is a "functional relationship" (p. 73) between associations, i.e. an association depends on another association. For example, an association may be a subset of another.

A constraint between associations is represented by a dotted arrow connecting one association to the other, labeled by the constraint name.





A ConstraintBetweenAssociations always consists of three parts: an independent Association modelled by the isIndependentIn relationship, a dependent Association modelled by the isDependentIn relationship, and a Predicate which expresses the constraint itself.

2.2 Integration: Object Model

The EER description in Figure 2 and the following collection of GRAL predicates summarize the previous ones and express the abstract syntax of the object model of OMT.

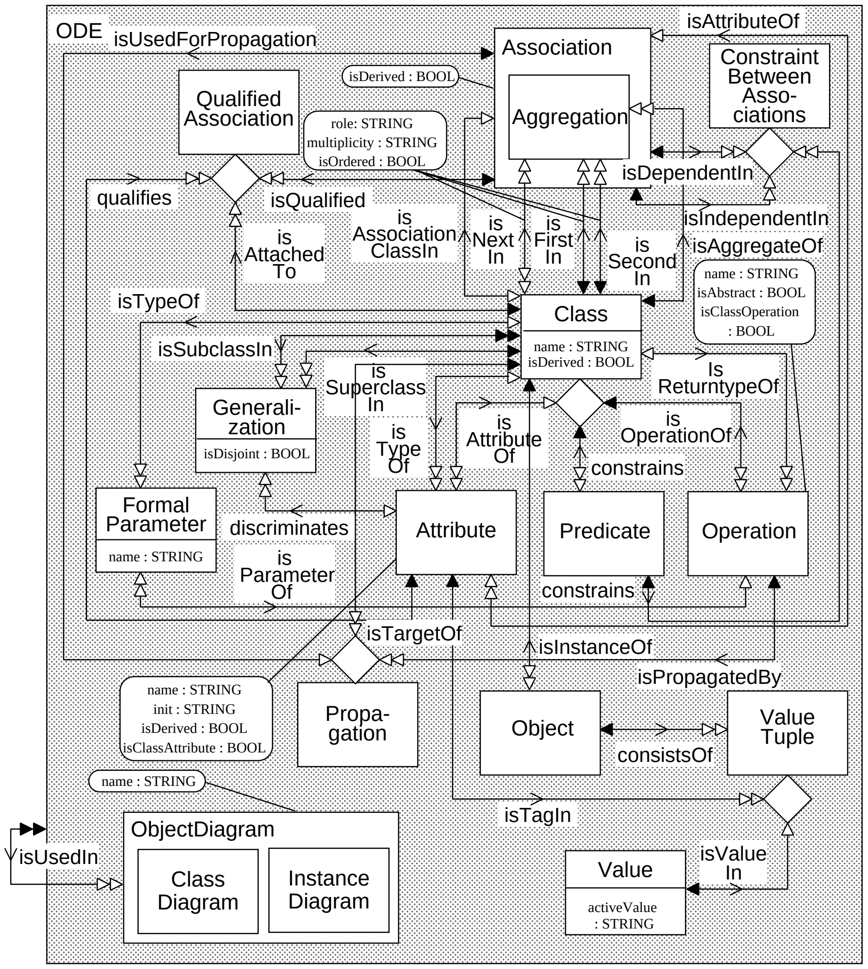


Figure 2: EER Diagram of the Object Model

For G **in** *ObjectModel* **assert**

OM1 : $\forall\, od_1, od_2 : V_{ObjectDiagram} \mid od_1 \neq od_2 \bullet od_1.name \neq od_2.name$;

OM2 : $\forall\, c_1, c_2 : V_{Class} \mid c_1 \neq c_2 \bullet c_1.name \neq c_2.name$;

OM3 : $\forall f : E_{isFirstIn};\ f_1 : E_{isSecondIn} \mid \omega(f) = \omega(f_1)$
$\bullet\, f.name \neq f_1.name$;

OM4 : $\forall\, q : V_{QualifiedAssociation};\ a : V_{Attribute};\ c : V_{Class} \mid$
$a \rightharpoonup_{qualifies} q \leftharpoonup_{isAttachedTo} c$
$\bullet\ (\exists\, c_1 : V_{Class};\ r : V_{Association} \mid r \rightharpoonup_{isQualified} q$
$\bullet\ (c \rightharpoonup_{isFirstIn} r \leftharpoonup_{isSecondIn} c_1 \leftharpoonup_{isAttributeOf} a) \vee$
$(c \rightharpoonup_{isSecondIn} r \leftharpoonup_{isFirstIn} c_1 \leftharpoonup_{isAttributeOf} a))\ ;$

OM5 : $\forall\, c : V_{Class}\ \bullet \neg\ (c\ (\rightarrow_{isSuperclassIn} \leftarrow_{isSubclassIn})^{+}\ c)\ ;$

OM6 : $\forall\, o : V_{Object};\ t : V_{ValueTuple} \mid o \rightharpoonup_{consistsOf} t$
$\bullet\ t \leftharpoonup_{isTagIn} \rightharpoonup_{isAttributeOf} \leftharpoonup_{isInstanceOf} o\ ;$

OM7 : $\forall\, p : V_{Operation};\ c : V_{Class} \mid p.isAbstract = TRUE\ \wedge$
$p \rightharpoonup_{isOperationOf} c$
$\bullet\ degree(\rightharpoonup_{isSuperclassIn}, c) > 0\ ;$

OM8 : $\forall\, p : V_{Operation};\ c : V_{Class} \mid p.isAbstract = TRUE\ \wedge$
$p \rightharpoonup_{isOperationOf} c$
$\bullet\ (\forall\, c_1 : V_{Class} \mid c\ (\rightharpoonup_{isSuperclassIn} \leftharpoonup_{isSubclassIn})^{+} c_1$
$\bullet\ (\exists\, p_1 : V_{Operation} \mid p_1 \rightharpoonup_{isOperationOf} c_1$
$\bullet\ p.name = p_1.name\ \wedge$
$p_1.isAbstract = FALSE))\ ;$

OM9 : $\forall\, p : V_{Operation} \mid p.isAbstract = TRUE$
$\bullet\ (\forall\, p_1 : V_{Operation} \mid p \rightharpoonup_{isOperationOf} (\rightharpoonup_{isSuperclassIn} \leftharpoonup_{isSubclassIn})^{+}$
$\leftharpoonup_{isOperationOf} p_1$
$\bullet\ p.name \neq p_1.name);$

OM10 : $\forall\, a : V_{Association} \mid degree(\leftharpoonup_{isAssociationClassIn}, a) > 0$
$\bullet\ degree(\leftharpoonup_{isNextIn}, a) = 0\ ;$

OM11 : $\forall\, g : V_{Generalization};\ a : V_{Attribute} \mid a \rightharpoonup_{discriminates} g$
$\bullet\ a \rightharpoonup_{isAttribute} \rightharpoonup_{isSuperclassIn} g\ ;$

OM12 : $\forall\, pr : V_{Propagation};\ p : V_{Operation};\ c : V_{Class} \mid$
$p \rightharpoonup_{isPropagated} pr \leftharpoonup_{isTargetOf} c$
$\bullet\ (\exists\, c_1 : V_{Class};\ a : V_{Association} \mid a \rightharpoonup_{isUsedForPropagation} pr$
$\bullet\ (c \rightharpoonup_{isFirstIn} a \leftharpoonup_{isSecondIn} c_1 \leftharpoonup_{isOperationOf} p) \vee$
$(c \rightharpoonup_{isSecondIn} a \leftharpoonup_{isFirstIn} c_1 \leftharpoonup_{isOperationOf} p))\ ;$

OM13 : $\forall\, pr : V_{Propagation};\ p : V_{Operation};\ c : V_{Class} \mid$
$p \rightharpoonup_{isPropagated} pr \leftharpoonup_{isTargetOf} c$
$\bullet\ (\exists\, p_1 : V_{Operation} \bullet p_1 \rightharpoonup_{isOperationOf} c \wedge p.name = p_1.name)\ ;$

OM14 : $\forall\, c : V_{Class} \mid c.isDerived = TRUE$
$\bullet\ \forall\, a : V_{Attributes} \mid c \leftharpoonup_{isAttributeOf} a \bullet a.isDerived = TRUE\ ;$

end.

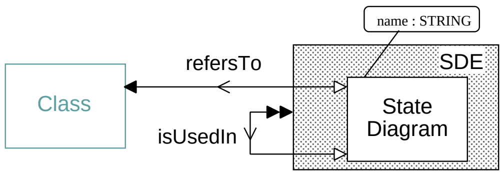
3 Dynamic Model

The *dynamic model* of a system describes the organization of control in the system, i.e. those aspects of a system "concerned with time and the sequencing of operations" (p. 18). The dynamic model is represented by a set of *state diagrams* which provide a formal graphical notation.

3.1 State Diagram

A *state diagram* is a named sheet of paper showing one or more symbols 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 diagram has no explicit notation in OMT. One may take the sheet of paper as the graphical representation.



A StateDiagram comprises all elements which are assigned to it by the isUsedIn relationship. Here, the abstract concept state diagram element, SDE for short, is used as a placeholder for these elements which have to be specialized concepts of the SDE. For graphical simplicity, this is done in Figure 3 (p. 29) for the first time. A state diagram always refersTo a given Class.

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

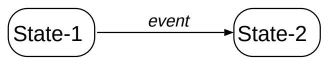
DM1 : $\forall\, sd_1, sd_2 : V_{StateDiagram} \mid sd_1 \neq sd_2 \bullet sd_1.name \neq sd_2.name$;

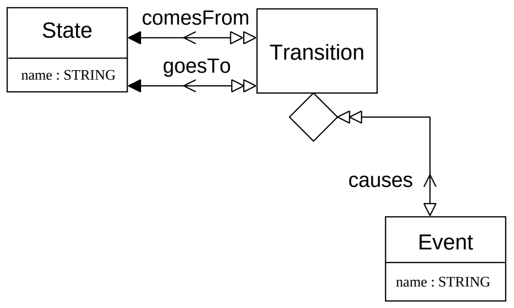
❈

Event causes Transition between States:

A *state* of an object denotes a period of time in which the object does not change its properties connected to this state. An *event* is a stimulus that causes a *transition* from one state to another.

A state is represented by a rounded rectangle labeled by its name. A transition is represented by a line with an arrow connecting two states. Events are annotated as text.





A Transition[3] 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.

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

$$\text{DM2}: \quad \forall\, s_1, s_2 : V_{State} \mid s_1 \neq s_2 \bullet s_1.name \neq s_2.name \;;$$

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

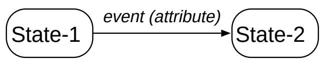
$$\text{DM3}: \quad \forall\, e_1, e_2 : V_{Event} \mid e_1.name = e_2.name \\ \bullet\; e_1 = e_2 \vee \neg\,(e_1 \rightharpoonup_{causes} \rightharpoonup_{comesFrom} \leftharpoondown_{comesFrom} \leftharpoondown_{causes} e_2) \;;$$

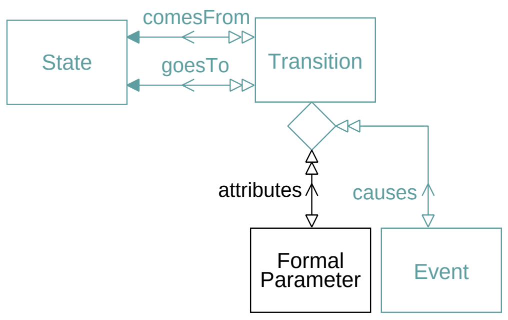
❊

Event with Attribute:

An event conveys information from one object to another. The "data values conveyed by an event are its *attributes*" (p. 85).

An attribute is annotated in parentheses after the event name.





[3] Here, the Transition is modelled by an aggregation because we will include more concepts into our modeling.

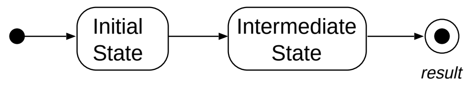
Attributes of events are modelled by FormalParameters which are directly assigned to Transitions by the attributes relationship and therefore indirectly to its Event.

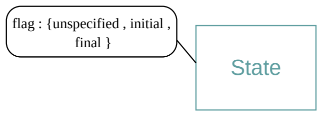
❈

Initial and Final States:

An *initial state* denotes the creation whereas a *final state* denotes the deletion of an object.

An initial state is represented by a bullet and an arrow pointing to this state. The final state is represented similarly but with an additional circle around the bullet.





Initial and final states are recognizable by the value of the attribute flag which may be unspecified, initial or final. Note that in the OMT initial and final states are not modelled equally. The 'bull's-eye', which represents a final state, is really a state whereas the bullet only marks an initial state and is not a state itself.

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

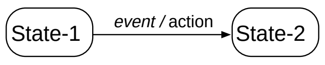
DM4 : $\forall\, s : V_{State} \mid s.flag = final \bullet degree(\leftharpoonup_{comesFrom}, s) = 0\ ;$

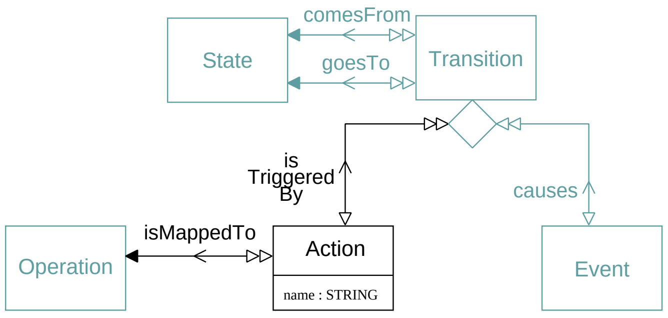
❈

Action on a Transition:

An *action* on a transition is an "instantaneous operation" (p. 92) which is triggered by an event.

The OMT notation for an action on a transition is a slash (/) and its name after the name of the event that causes it.





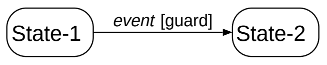
An Action on a transition isTriggeredBy an event which is connected to this transition. An action isMappedTo an Operation of the object model.

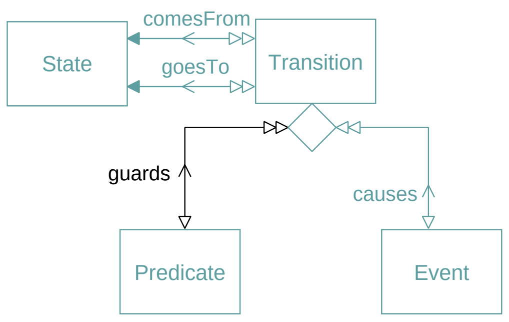
❈

Guarded Transition:

A *guarded transition* is a transition with an additional condition on its event. A guarded transition "fires when its event occurs, but only if the guard condition is true" (p. 91).

The condition is annotated in brackets following the event name.





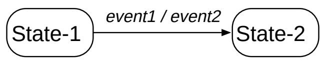
The condition is modelled by a Predicate which guards the transition.

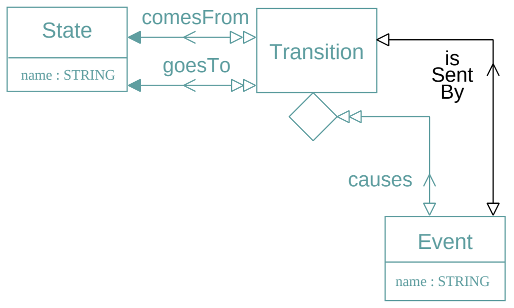
❈

Output Event on a Transition:

An object "can perfom the action of sending an event to another object" (p. 103).

The OMT notation for an event which is sent to another object is a slash (/) and its name after the name of the event that causes it. Event names are shown in italics which make them distinguishable from action names.





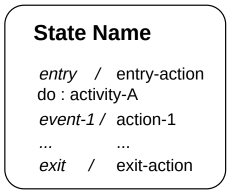
An Event which is sent to another object due to a Transition is modelled by the isSentBy relationship.

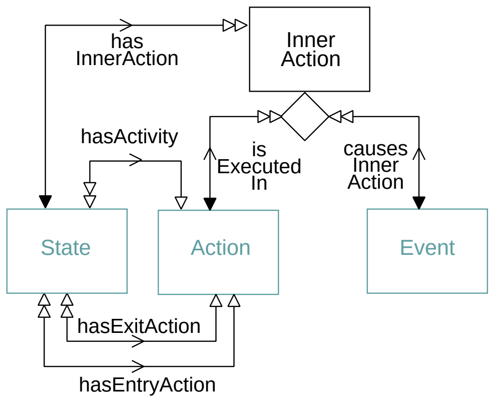
❈

Actions and Activity while in a State:

A state is a period of time in which an *activity* can be executed, and at its beginning and at its end an *entry-* and an *exit-action* may occur. Additionally, several *actions* can be triggered by events during this time.

The actions are listed within the rectangle, which represents the state, distinguished by the keywords 'entry', 'do' and 'exit'. Furthermore, the actions caused by events are listed in the same manner as event and action on a transition.



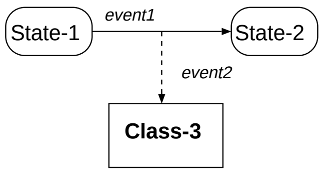


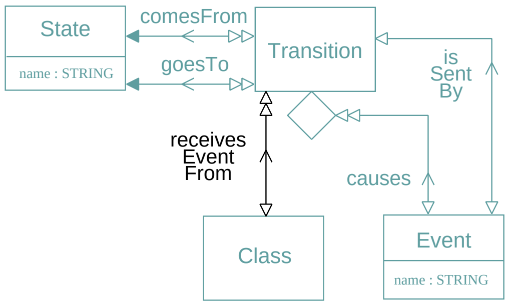
A State can be connected to an Action in the three different ways described above: the hasActivity relationship denotes the do-action, hasEntryAction denotes the entry-action, and hasExitAction denotes the exit-action. The InnerActions comprise Events and Actions which are assigned to them by the isExecutedIn and causesInnerAction relationships.

Sending an event to another object:

Events can be explictly *sent* to a set of *objects* which are described by their class.

The OMT notation for sending an event to a class is a dotted arrow from the transition to the class labeled by this event.





The sending event is already modelled by the isSentBy relationship whereas the receiving Class is now modelled by the receivesEventFrom relationship.

Constraint A transition can only explicitly send events to a class.

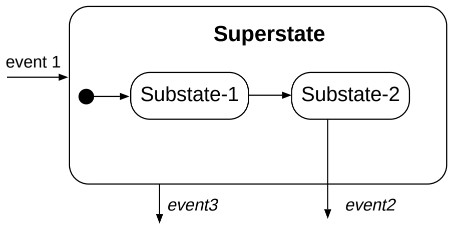
$$\text{DM5}: \quad \forall\, t : V_{Transition} \mid degree(\leftharpoonup_{receivesEventFrom}, t) = 1 \bullet degree(\leftharpoonup_{isSentBy}, t) = 1 \;;$$

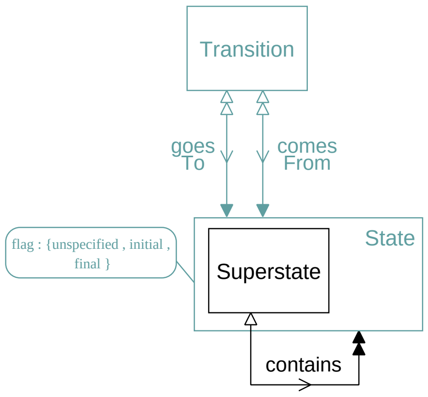
❋

State Generalization (Nesting)

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

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





A Superstate may contain one or more other States.

There are several constraints connected to superstates.

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

$$\text{DM6}: \quad \forall\, su : V_{Superstate} \bullet \#\{s : V_{State} \mid su \rightharpoonup_{contains} s \wedge s.flag = initial\} \leq 1 \;;$$

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

$$\text{DM7}: \quad \forall\, su : V_{Superstate} \mid degree(\leftharpoonup_{goesTo}, su) > 0 \bullet (\exists_1\, s : V_{State} \bullet su \rightharpoonup_{contains} s \wedge s.flag = initial\ ) \;;$$

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

$$\text{DM8}: \quad isForest(eGraph(\rightharpoonup_{contains})) \;;$$

Constraint On the top of state diagrams there must exist one initial state.

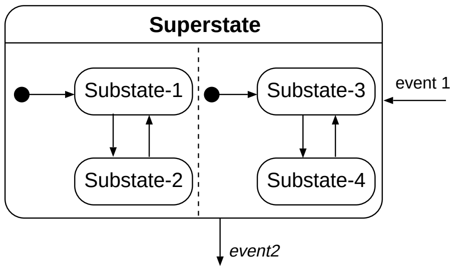
$$\text{DM9}: \quad \forall\, sd : V_{StateDiagram} \bullet (\exists_1\, s : V_{State} \bullet s \rightharpoonup_{isUsedIn} sd \land \mathrm{degree}(\leftharpoonup_{contains}, s) = 0 \land s.flag = initial) ;$$

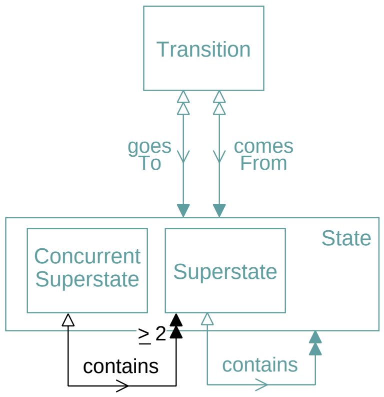
❄

Concurrent Subdiagrams:

A *concurrent subdiagram* describes "concurrency within the state of a single object" in which the "state of the object comprises one state from each subdiagram" (p. 99).

The OMT notation for subdiagrams is similar to normal state diagram symbols in which the subdiagrams are distinguished by dotted lines. The name of the overall composite state can be written at the top, separated by a solid line.





Concurrent subdiagrams are modelled by ConcurrentSuperstates which contain two or more Superstates that make the concurrent states distinguishable.

Constraint Transitions are not allowed between states which belong to different superstates of one concurrent superstate.

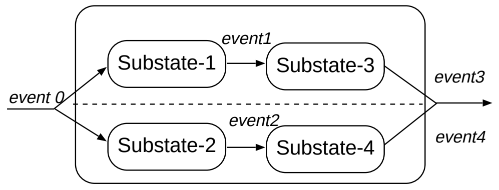
$$\text{DM10}: \quad \forall\, s : V_{State};\ su : V_{Superstate};\ csu : V_{ConcurrentSuperstate} \mid csu \rightharpoonup_{contains} su (\rightharpoonup_{contains})^{+} s \bullet (\forall\, s_1 : V_{State} \mid s_1 \leftharpoonup_{goesTo} \rightharpoonup_{comesFrom} s \lor s_1 \leftharpoonup_{comesFrom} \rightharpoonup_{goesTo} s \bullet su (\rightharpoonup_{contains})^{+} s_1) ;$$

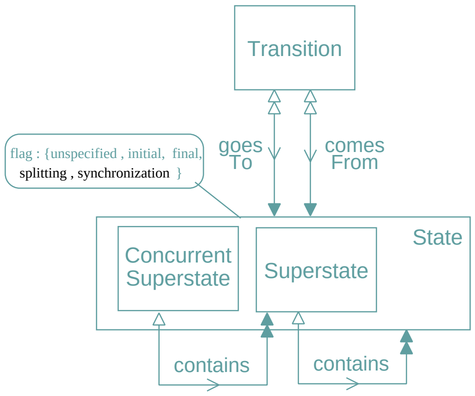
❄

Splitting and Synchronization of control:

Splitting of control is used when a transition branches from a single state into several states of a concurrent superstate. I.e. an object is simultaneously in two or more substates. *Synchronization of control* is the counterpart of splitting. It is used when an object returns from two or more concurrent states to one single state back and this transition needs to be synchronized.

The OMT notation for splitting of control is an arrow that forks and for synchronization of control an arrow with a forked tail.





Splitting and synchronization of control is modelled by a State which is used for splitting or synchronization of control.

Constraint States which are used to split and to synchronize control are not allowed to contain other states.

$$\text{DM11}: \quad \forall\, s : V_{State} \mid s.flag = splitting \vee s.flag = synchronization$$
$$\bullet\ degree(\rightharpoonup_{contains}, s) = 0\ ;$$

Constraint The outgoing transitions from a state to be used to split control must end in states which belong to the same concurrent superstate.

$$\text{DM12}: \quad \forall\, s : V_{State} \mid s.flag = splitting$$
$$\bullet\ (\forall\, s_1 : V_{State} \mid s \leftharpoonup_{comesFrom} \rightharpoonup_{goesTo}\ s_1$$
$$\bullet\ (\exists_1\, csu : V_{ConcurrentSuperstate} \bullet csu(\rightharpoonup_{contains})^{+} s_1))\ ;$$

Constraint The incoming transitions to state to be used to synchronize control must end in states which belong to the same concurrent superstate.

$$\text{DM13}: \quad \forall\, s : V_{State} \mid s.flag = synchronization$$
$$\bullet\ (\forall\, s_1 : V_{State} \mid s_1 \leftharpoonup_{comesFrom} \rightharpoonup_{goesTo}\ s$$
$$\bullet\ (\exists_1\, csu : V_{ConcurrentSuperstate} \bullet csu(\rightharpoonup_{contains})^{+} s_1))\ ;$$

3.2 Integration: Dynamic Model

The EER description in Figure 3 and the following collection of GRAL predicates summarize the previous ones and express the abstract syntax of the dynamic model of OMT. Furthermore, two constraints are added.

Constraint An event which is mapped to an operation, must belong to a state diagram which refers to the class of the operation.

DM14 : $\forall\, e : V_{Event};\ o : V_{Operation} \mid e \rightharpoonup_{mapsTo} o$
$\bullet\ e \rightharpoonup_{isUsedIn} \rightharpoonup_{refersTo} \leftharpoonup_{isOperationOf}\ o$;

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

DM15 : $\forall\, a : V_{Action} \bullet degree(\rightharpoonup_{isMappedTo}, a) + degree(\rightharpoonup_{triggers}, a) = 1$;

❄

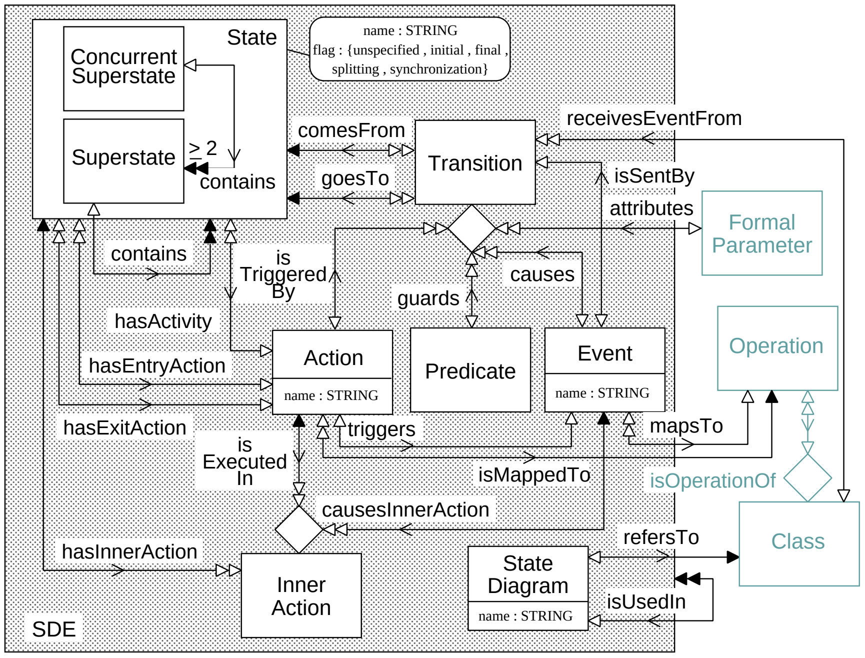


Figure 3: EER Diagram of the Dynamic Model

For G **in** *DynamicModel* **assert**

DM1 : $\forall\, sd_1, sd_2 : V_{StateDiagram} \mid sd_1 \neq sd_2 \bullet sd_1.name \neq sd_2.name$;

DM2 : $\forall\, s_1, s_2 : V_{State} \mid s_1 \neq s_2 \bullet s_1.name \neq s_2.name$;

DM3 : $\forall\, e_1, e_2 : V_{Event} \mid e_1.name = e_2.name$
$\bullet\ e_1 = e_2 \vee \neg\,(e_1 \rightharpoonup_{causes} \rightharpoonup_{comesFrom} \leftharpoonup_{comesFrom} \leftharpoonup_{causes} e_2)$;

DM4 : $\forall\, s : V_{State} \mid s.\mathit{flag} = \mathit{final} \bullet \mathit{degree}(\leftharpoonup_{comesFrom}, s) = 0\ ;$

DM5 : $$\begin{aligned}&\forall\, t : V_{Transaction} \mid \mathit{degree}(\leftharpoonup_{receivesEventFrom}, t) = 1\\ &\quad \bullet\ \mathit{degree}(\leftharpoonup_{isSentBy}, t) = 1\ ;\end{aligned}$$

DM6 : $$\begin{aligned}&\forall\, su : V_{Superstate}\\ &\quad \bullet\ \#\{s : V_{State} \mid su \rightharpoonup_{contains} s \wedge s.\mathit{flag} = \mathit{initial}\} \leq 1\ ;\end{aligned}$$

DM7 : $$\begin{aligned}&\forall\, su : V_{Superstate} \mid \mathit{degree}(\leftharpoonup_{goesTo}, su) > 0\\ &\quad \bullet\ (\exists_1\, s : V_{State} \bullet su \rightharpoonup_{contains} s \wedge s.\mathit{flag} = \mathit{initial}\ )\ ;\end{aligned}$$

DM8 : $\mathit{isForest}(\mathit{eGraph}(\rightharpoonup_{contains}))\ ;$

DM9 : $$\begin{aligned}&\forall\, sd : V_{StateDiagram}\\ &\quad \bullet\ (\exists_1\, s : V_{State} \bullet s \rightharpoonup_{isUsedIn} sd \wedge \mathrm{degree}(\leftharpoonup_{contains}, s) = 0 \wedge\\ &\qquad\qquad\qquad s.\mathit{flag} = \mathit{initial})\ ;\end{aligned}$$

DM10 : $$\begin{aligned}&\forall\, s : V_{State};\ su : V_{Superstate};\ csu : V_{ConcurrentSuperstate} \mid\\ &\qquad csu \rightharpoonup_{contains} su(\rightharpoonup_{contains})^{+} s\\ &\quad \bullet\ (\forall\, s_1 : V_{State} \mid s_1 \leftharpoonup_{goesTo}\rightharpoonup_{comesFrom} s \vee\\ &\qquad\qquad s_1 \leftharpoonup_{comesFrom}\rightharpoonup_{goesTo} s\\ &\qquad \bullet\ su(\rightharpoonup_{contains})^{+} s_1\ )\ ;\end{aligned}$$

DM11 : $$\begin{aligned}&\forall\, s : V_{State} \mid s.\mathit{flag} = \mathit{splitting} \vee s.\mathit{flag} = \mathit{synchronization}\\ &\quad \bullet\ \mathit{degree}(\rightharpoonup_{contains}, s) = 0\ ;\end{aligned}$$

DM12 : $$\begin{aligned}&\forall\, s : V_{State} \mid s.\mathit{flag} = \mathit{splitting}\\ &\quad \bullet\ (\forall\, s_1 : V_{State} \mid s \leftharpoonup_{comesFrom}\rightharpoonup_{goesTo} s_1\\ &\qquad \bullet\ (\exists_1\, csu : V_{ConcurrentSuperstate} \bullet csu(\rightharpoonup_{contains})^{+} s_1))\ ;\end{aligned}$$

DM13 : $$\begin{aligned}&\forall\, s : V_{State} \mid s.\mathit{flag} = \mathit{synchronization}\\ &\quad \bullet\ (\forall\, s_1 : V_{State} \mid s_1 \leftharpoonup_{comesFrom}\rightharpoonup_{goesTo} s\\ &\qquad \bullet\ (\exists_1\, csu : V_{ConcurrentSuperstate} \bullet csu(\rightharpoonup_{contains})^{+} s_1))\ ;\end{aligned}$$

DM14 : $$\begin{aligned}&\forall\, e : V_{Event};\ o : V_{Operation} \mid e \rightharpoonup_{mapsTo} o\\ &\quad \bullet\ e \rightharpoonup_{isUsedIn}\rightharpoonup_{refersTo}\leftharpoonup_{isOperationOf} o\ ;\end{aligned}$$

DM15 : $\forall\, a : V_{Action} \bullet \mathit{degree}(\rightharpoonup_{isMappedTo}, a) + \mathit{degree}(\rightharpoonup_{triggers}, a) = 1\ ;$

end.

4 Functional Model

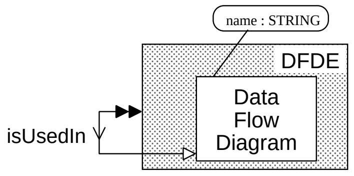
In OMT the *functional model* of a system describes the computation of values in the system, i.e. those aspects of a system "concerned with transformations of values – functions, mappings, constraints, and functional dependencies" (p. 18).

The functional model is represented by a set of *data flow diagrams* which provide a formal graphical notation.

4.1 Data Flow Diagram

A *data flow diagram* is a named sheet of paper showing one or more symbols which represent the possible "flow of data values from their sources in objects ... to their destination" (p. 124).

A data flow diagram has no explicit notation in OMT. One may take the sheet of paper as the graphical representation.



A DataFlowDiagram comprises all elements which are assigned to it by the isUsedIn relationship. Here, the abstract concept data flow diagram element, DFDE for short, is used as a placeholder for these elements which have to be specialized concepts of the DFDE. For graphical simplicity, this is done in Figure 4 (p. 39) for the first time.

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

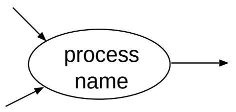
FM1 : $\forall\, dfd_1, dfd_2 : V_{DataFlowDiagram} \mid dfd_1 \neq dfd_2$
$\bullet\ dfd_1.name \neq dfd_2.name$;

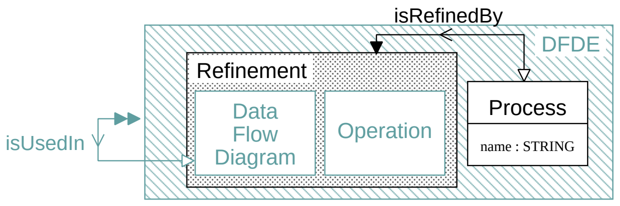
❋

Process:

A *process* describes the transformation of input values to output values. The "lowest-level processes are pure functions" (p. 124), but higher-level processes can be refined by entire data flow diagrams. Strictly speaking, the process is refined by the elements of these data flow diagrams.

The OMT symbol for a process is an ellipse containing its name. Its input values are represented by incoming arrows and the output values by outgoing arrows.





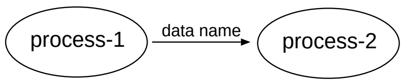
A process is modelled by the corresponding concept Process. Processes can be refined either by additional DataFlowDiagrams (this case expresses higher-level processes) or by Operations which describe lowest-level processes. These Refinements are assigned to processes by the isRefinedBy relationship.

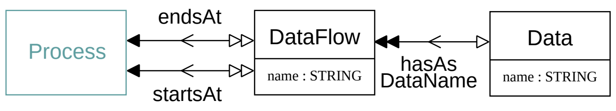
❈

Data Flow between Processes:

A *data flow* "connects the output of an object or process to the input of another object or process" (p. 126). Data flows are objects which are in motion in the system.

The OMT notation for a data flow is an arrow labeled by its name between the producer and the consumer of the data value.





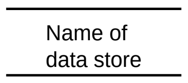
A DataFlow starts at one producer and ends at one consumer which is expressed by the startsAt and the endsAt relationships. Additionally, the data name is expressed by the Data which is connected by the hasAsDataName relationship.

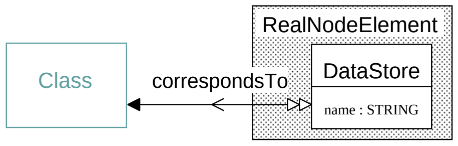
❈

Data Store or File Object:

A *data store* "is a passive object within a data flow diagram that stores data for later access" which means that it "does not generate operations on its own" (p. 127). A data store keeps objects which are not in motion in a system.

The OMT notation for a data store is a pair of parallel lines containing its name.





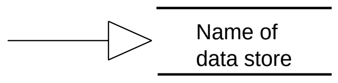
A data store is modelled by the corresponding concept DataStore. The abstract concept RealNodeElement is used to simplify the metamodel in the further modeling. A data store correspondsTo a Class in the object model.

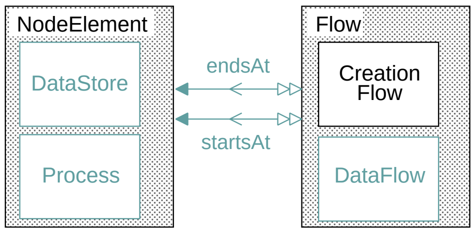
❈

Data Flow that results in a Data Store:

In OMT a data flow cannot only express the flow of an object but also the creation of an object. The *result* of a data flow, the created object, is stored *in a data store.*

The OMT notation for a data flow which creates objects is a line with a hollow triangle at its end pointing to a data store.





A data flow which creates an object is modelled by a CreationFlow. It is similar to a normal DataFlow with respect to their relationships, so the generalization is used to model this aspect. The abstract concept Flow is an artificial concept which is introduced to make the metamodel simpler. The same applies to the abstract concept NodeElement.

Constraint A data flow which creates an object always ends in a data store.

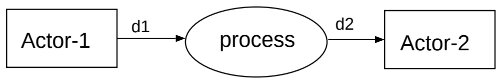
$$\text{FM2}: \quad \forall\, cf : V_{CreationFlow} \bullet cf \rightharpoonup_{endsAt} \subseteq V_{DataStore} \;;$$

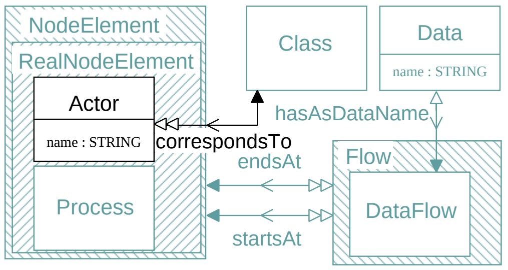
❈

Actor Objects (as Source or Sink of Data):

An *actor* "is an active object that drives the data flow graph by producing or consuming values" (p. 126). An actor is not controlled by the system and can independently produce data.

An actor is drawn as a rectangle and its name within it.





An Actor can produce or consume objects, expressed by DataFlows and Data, in the same way Processes do, so we use the generalization concept NodeElement to model this aspect. Additionally, RealNodeElement is used to simplify the following constraint to be preserved. An actor correspondsTo a Class in the object model.

Constraint Either the producer or the consumer of a value represented by a data flow must be a process.

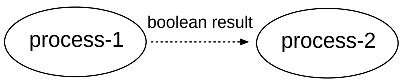
$$\mathsf{FM3}: \quad \forall\, v, w : V_{RealNodeElement};\ d : V_{DataFlow} \mid v \leftharpoonup_{startsAt} d \rightharpoonup_{endsAt} w \\ \bullet\ type(v) = Process \lor\ type(w) = Process\ ;$$

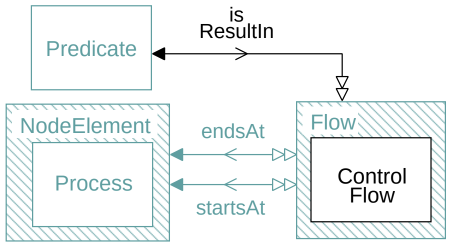
❈

Control Flow:

An *control flow* "is a Boolean value that affects whether a process is evaluated" (p. 129). A control flow is also a part of the dynamic model.

The OMT notation for a control flow is a dotted arrow labeled by a boolean result.





A control flow is modelled by the concept ControlFlow. It is similar to Flows with respect to their relationships. But there is a constraint to be preserved.

Constraint Control flows are only allowed between processes.

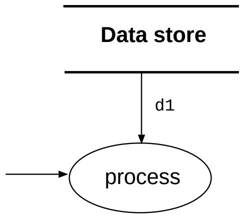
$$\mathsf{FM4}: \quad \forall\, c : V_{ControlFlow} \bullet c \rightharpoonup_{startsAt} \cup\ c \rightharpoonup_{endsAt} \subseteq V_{Process}\ ;$$

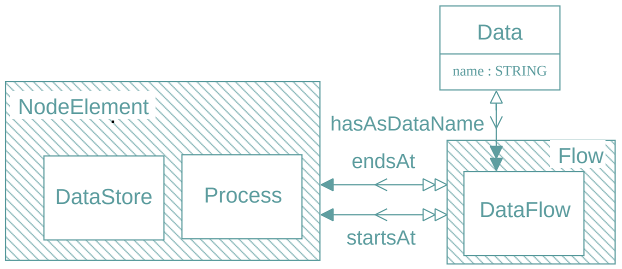
❈

Access of Data Store Value:

An *access of data store value*, which can only be made by processes, means that the values are moved from the data store to the process.

Access of data store value is represented by an arrow from the process to the data store. If only a part of the value is accessed, the part is annotated at the arrow.





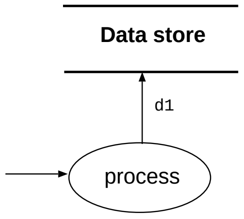
We already modelled the access of data store value by the startsAt and endsAt relationships. In this case the startsAt relationship is connected to the Data Store and the endsAt relationship is connected to the Process.

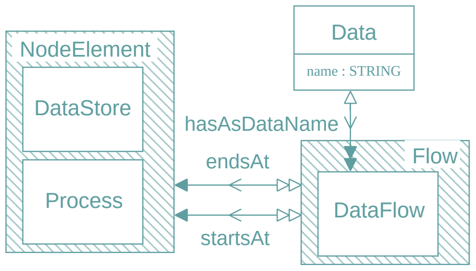
❋

Update of Data Store Value:

An *update of data store value*, which can only be made by processes, means that the values are moved from the process to the data store.

Update of data store values is represented by an arrow from the data store to the process. If only a part of the value is updated, the part is annotated at the arrow.





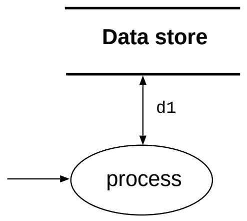
We already modelled the update of data store value by the startsAt and endsAt relationships. In this case the startsAt relationship is connected to the Process and the endsAt relationship is connected to the DataStore.

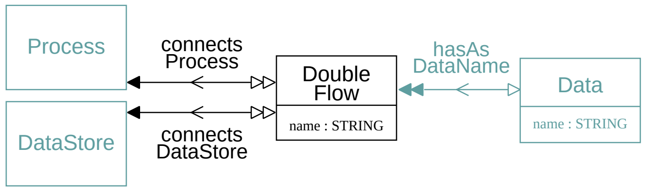
❋

Access and Update of Data Store Value:

An *access and update of data store value*, which can only be made by processes, means that the values are moved from the data store to the process, updated and moved back.

Access and update of data store value is represented by a double-headed arrow between the data store and the process. If only a part of the value is updated, the part is annotated at the arrow.





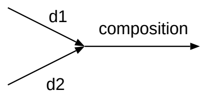
We modelled the access and the update of data store value by the concept DoubleFlow. It can only be connected to Processes and DataStores. We think that access and update is meant as a coherent concept and not as an abbreviation for a single access and a single update concept.

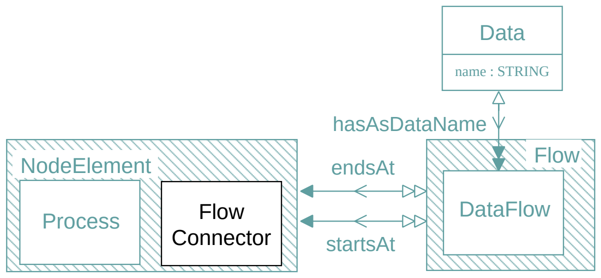
❈

Composition of Data Value:

Composition of data value means that several data flows are merged into one single data flow which represent its composition.

Composition of a data value is represented by a fork in which several incoming arrows are merged at one point to one outgoing arrow. The name of the composition is labeled at the outgoing arrow.





Composition of data value is modelled by the concept FlowConnector which serves as the connection point to the incoming and the outgoing DataFlows.

Constraint In a composition there are allowed one outgoing data flow and several incoming data flows.

$$\text{FM5}: \quad \forall\, fc : V_{FlowConnector} \mid degree(\leftharpoonup_{startsAt}, fc) = 1 \bullet degree(\leftharpoonup_{endsAt}, fc) \geq 2\ ;$$

Constraint Either the producer or the consumer of a composite value must be a process.

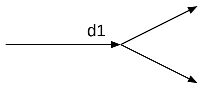
$$\text{FM6}: \quad \forall\, v, w : V_{NodeElement};\ d_1;\ d_2 : V_{DataFlow};\ fc : V_{FlowConnector} \mid v \leftharpoonup_{startsAt} d_1 \rightharpoonup_{endsAt} fc \leftharpoonup_{startsAt} d_2 \rightharpoonup_{endsAt} w \bullet type(v) = Process \vee type(w) = Process\ ;$$

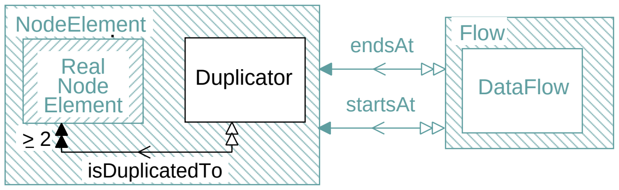
❈

Duplication of Data Value:

Duplication of data value means that a data flow is *duplicated* to several data flows.

Duplication of data value is represented by fork in which one incoming arrow is split at one point to several outgoing arrow.





Duplication of data value is modelled by the concept Duplicator which has an incoming DataFlow represented by the endsAt relationship and isDuplicatedTo the RealNodeElement (process, data store or actor) .

Constraint A duplicator has no outgoing data flows.

$$\text{FM7}: \quad \forall\, dp : V_{Duplicator} \bullet degree(\leftharpoondown_{startsAt}, dp) = 0 \;;$$

Constraint Either the producer or the consumer of a duplicated value must be a process.

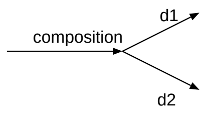
$$\text{FM8}: \quad \forall\, v : V_{NodeElement};\; w : V_{RealNodeElement};\; d : V_{DataFlow};\; dp : V_{Duplicator} \mid v \leftharpoondown_{startsAt} d \rightharpoonup_{endsAt} dp \rightharpoonup_{isDuplicatedTo} w \bullet type(v) = Process \vee type(w) = Process \;;$$

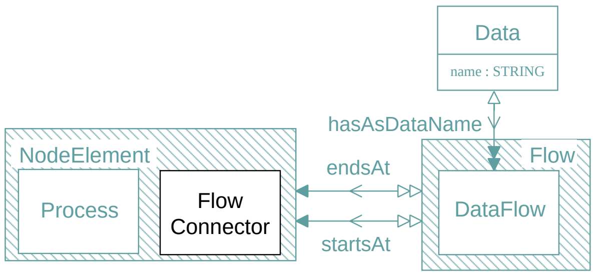
❋

Decomposition of Data Value:

Decomposition of data value means that one data flow is split into several data flows.

Decomposition of a data value is represented by a kind of fork in which one incoming arrow is split at one point to several outgoing arrows. The name of the composition is labeled at the incoming arrow.





Decomposition of data value is modelled by the concept FlowConnector which serves as the connection point to the incoming and outgoing DataFlows.

Constraint In a decomposition is only allowed one incoming data flow and several outgoing data flows.

FM9 : $\forall fc : V_{FlowConnector} \mid degree(\leftharpoondown_{endsAt}, fc) = 1$
$\bullet\ degree(\leftharpoondown_{startsAt}, fc) \geq 2$;

Constraint Flow connectors are only used in composition or decomposition.

FM10 : $\forall fc : V_{FlowConnector}$
$\bullet\ degree(\leftharpoondown_{startsAt}, fc) = 1 \vee degree(\leftharpoondown_{endsAt}, fc) = 1$;

❋

4.2 Integration: Functional Model

The EER description in Figure 4 and the following collection of GRAL predicates summarize the previous ones and express the abstract syntax of the functional model of OMT.

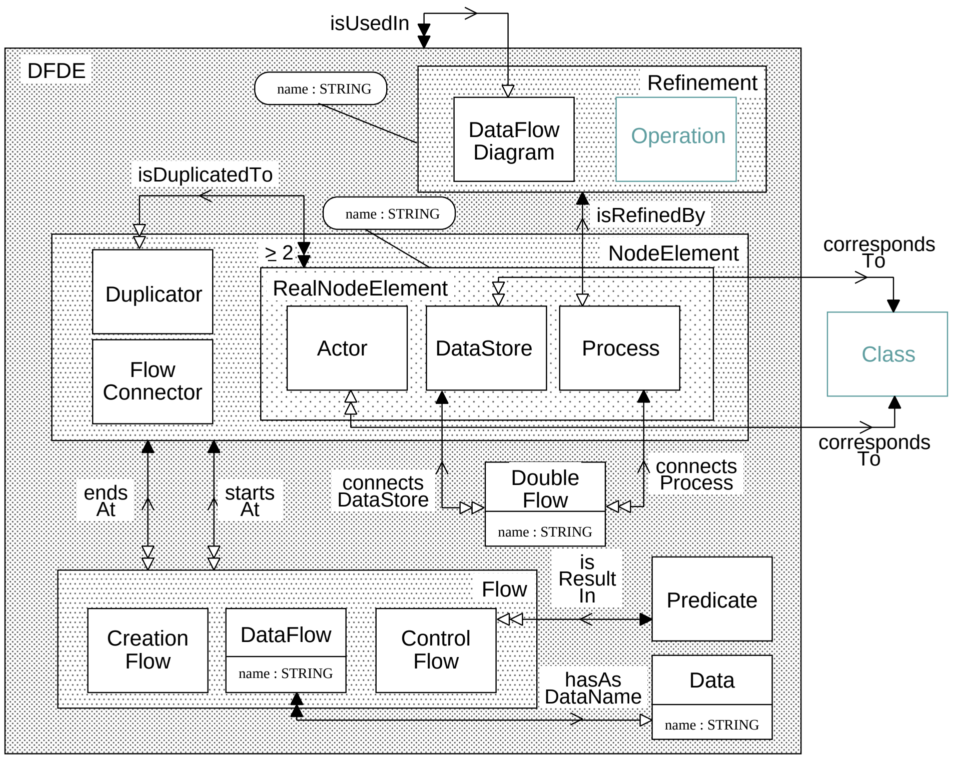


Figure 4: EER Diagram of the Functional Model

For G **in** *FunctionalModel* **assert**

FM1 : $\forall\, dfd_1, dfd_2 : V_{DataFlowDiagram} \mid dfd_1 \neq dfd_2 \bullet dfd_1.name \neq dfd_2.name\ ;$

FM2 : $\forall\, cf : V_{CreationFlow} \bullet cf \rightharpoonup_{endsAt} \subseteq V_{DataStore}\ ;$

FM3 : $\forall\, v, w : V_{RealNodeElement};\ d : V_{DataFlow} \mid v \leftharpoondown_{startsAt} d \rightharpoonup_{endsAt} w \bullet type(v) = Process \vee type(w) = Process\ ;$

FM4 : $\forall\, c : V_{ControlFlow} \bullet c \rightharpoonup_{startsAt} \cup c \rightharpoonup_{endsAt} \subseteq V_{Process}\ ;$

FM5 : $\forall\, fc : V_{FlowConnector} \mid degree(\leftharpoondown_{startsAt}, fc) = 1 \bullet degree(\leftharpoondown_{endsAt}, fc) \geq 2\ ;$

FM6 : $\forall\, v, w : V_{NodeElement};\ d_1;\ d_2 : V_{DataFlow};\ fc : V_{FlowConnector} \mid$
$v \leftharpoonup_{startsAt} d_1 \rightharpoonup_{endsAt} fc \leftharpoonup_{startsAt} d_2 \rightharpoonup_{endsAt} w$
$\bullet\ type(v) = Process \vee\ type(w) = Process\ ;$

FM7 : $\forall\, dp : V_{Duplicator} \bullet degree(\leftharpoonup_{startsAt}, dp) = 0\ ;$

FM8 : $\forall\, v : V_{NodeElement};\ w : V_{RealNodeElement};\ d : V_{DataFlow};\ dp : V_{Duplicator} \mid$
$v \leftharpoonup_{startsAt} d \rightharpoonup_{endsAt} dp \rightharpoonup_{isDuplicatedTo} w$
$\bullet\ type(v) = Process \vee\ type(w) = Process\ ;$

FM9 : $\forall fc : V_{FlowConnector} \mid degree(\leftharpoonup_{endsAt}, fc) = 1$
$\bullet\ degree(\leftharpoonup_{startsAt}, fc) \geq 2\ ;$

FM10 : $\forall fc : V_{FlowConnector}$
$\bullet\ degree(\leftharpoonup_{startsAt}, fc) = 1 \vee degree(\leftharpoonup_{endsAt}, fc) = 1\ ;$

end.

References

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

[CEW94] Carstensen, Martin; Ebert, Jürgen; Winter, Andreas; *Entity-Relationship-Diagramme und Graphklassen.* in german
Koblenz: Universität Koblenz-Landau, Technical Report, 1994.

[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, p. 163-178. Berlin: Springer, 1996.

[F97] Franzke, Angelika; *GRAL 2.0: A Reference Manual.* Koblenz: Universität Koblenz-Landau, Fachbericht Informatik 3/97, 1997.

[RBP+91] Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; Lorensen, William; *Object-Oriented Modeling and Design.* Englewood Cliffs: Prentice-Hall, 1991.

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

[SE97] Süttenbach, Roger; Ebert, Jürgen; *A Booch Metamodel.* Koblenz: Universität Koblenz-Landau, Fachbericht Informatik 5/97, 1997.

A The Overall Metamodel

The EER description in Figure 5 and the following collection of GRAL predicates summarize the abstract syntax of the object, the dynamic and the functional model, and express the metamodel of OMT. Equal concepts which are used in several models are merged into one concept and are assigned to that model in which they are appeared at first.

The *object model* which comprises

- class diagrams

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

The *dynamic model* which comprises

- state diagrams,

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

The *funtional model* which comprises

- data flow diagrams

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

The *integration* between these models is expressed by merging entity types and by relationships between these views, for example the mapsTo relationship between the Event of the dynamic model and the Operation of the object model.

The *metamodel* of OMT consists of

- 26 entity types,
- 64 relationship types,
- 33 attributes,
- 10 generalizations,
- 7 aggregations, and
- 39 GRAL predicates

in its entirety.

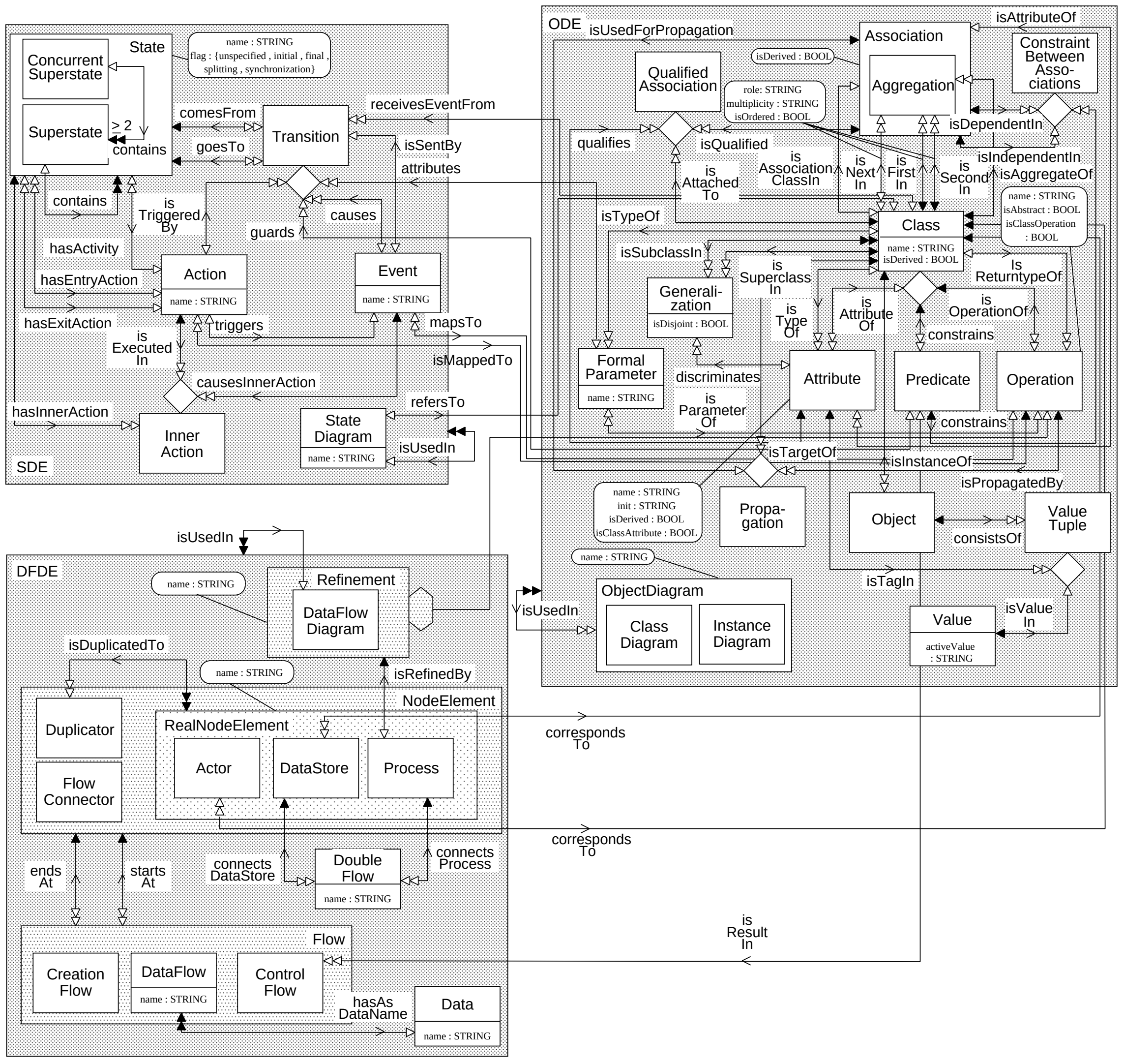


Figure 5: Metamodel of the Object Modeling Technique (OMT)

For G **in** OMT **assert**

OM1 : $\forall\, od_1, od_2 : V_{ObjectDiagram} \mid od_1 \neq od_2 \bullet od_1.name \neq od_2.name$;

OM2 : $\forall\, c_1, c_2 : V_{Class} \mid c_1 \neq c_2 \bullet c_1.name \neq c_2.name$;

OM3 : $\forall f : E_{isFirstIn};\ f_1 : E_{isSecondIn} \mid \omega(f) = \omega(f_1)$
$\bullet\, f.name \neq f_1.name$;

OM4 : $\forall\, q : V_{QualifiedAssociation};\ a : V_{Attribute};\ c : V_{Class} \mid$
$a \rightharpoonup_{qualifies} q \leftharpoonup_{isAttachedTo} c$
$\bullet\, (\exists\, c_1 : V_{Class};\ r : V_{Association} \mid r \rightharpoonup_{isQualified} q$
$\bullet\, (c \rightharpoonup_{isFirstIn} r \leftharpoonup_{isSecondIn} c_1 \leftharpoonup_{isAttributeOf} a) \vee$
$(c \rightharpoonup_{isSecondIn} r \leftharpoonup_{isFirstIn} c_1 \leftharpoonup_{isAttributeOf} a))$;

OM5 : $\forall\, c : V_{Class} \;\bullet \neg\, (c\ (\rightharpoonup_{isSuperclassIn} \leftharpoonup_{isSubclassIn})^{+}\ c)\ ;$

OM6 : $\forall\, o : V_{Object};\ t : V_{ValueTuple} \mid o \rightharpoonup_{consistsOf} t$
$\quad \bullet\ t \leftharpoonup_{isTagIn} \rightharpoonup_{isAttributeOf} \leftharpoonup_{isInstanceOf}\ o\ ;$

OM7 : $\forall\, p : V_{Operation};\ c : V_{Class} \mid p.isAbstract = TRUE \wedge$
$\qquad p \rightharpoonup_{isOperationOf} c$
$\quad \bullet\ degree(\rightharpoonup_{isSuperclassIn}, c) > 0\ ;$

OM8 : $\forall\, p : V_{Operation};\ c : V_{Class} \mid p.isAbstract = TRUE \wedge$
$\qquad p \rightharpoonup_{isOperationOf} c$
$\quad \bullet\ (\forall\, c_1 : V_{Class} \mid c\ (\rightharpoonup_{isSuperclassIn} \leftharpoonup_{isSubclassIn})^{+} c_1$
$\qquad \bullet\ (\exists\, p_1 : V_{Operation} \mid p_1 \rightharpoonup_{isOperationOf} c_1$
$\qquad\quad \bullet\ p.name = p_1.name \wedge$
$\qquad\quad p_1.isAbstract = FALSE))\ ;$

OM9 : $\forall\, p : V_{Operation} \mid p.isAbstract = TRUE$
$\quad \bullet\ (\forall\, p_1 : V_{Operation} \mid p \rightharpoonup_{isOperationOf} (\rightharpoonup_{isSuperclassIn} \leftharpoonup_{isSubclassIn})^{+}$
$\qquad \leftharpoonup_{isOperationOf} p_1$
$\qquad \bullet\ p.name \neq p_1.name);$

OM10 : $\forall\, a : V_{Association} \mid degree(\leftharpoonup_{isAssociationClassIn}, a) > 0$
$\quad \bullet\ degree(\leftharpoonup_{isNextIn}, a) = 0\ ;$

OM11 : $\forall\, g : V_{Generalization};\ a : V_{Attribute} \mid a \rightharpoonup_{discriminates} g$
$\quad \bullet\ a \rightharpoonup_{isAttribute} \rightharpoonup_{isSuperclassIn} g\ ;$

OM12 : $\forall\, pr : V_{Propagation};\ p : V_{Operation};\ c : V_{Class} \mid$
$\quad p \rightharpoonup_{isPropagated} pr \leftharpoonup_{isTargetOf} c$
$\qquad \bullet\ (\exists\, c_1 : V_{Class};\ a : V_{Association} \mid a \rightharpoonup_{isUsedForPropagation} pr$
$\qquad\quad \bullet\ (c \rightharpoonup_{isFirstIn} a \leftharpoonup_{isSecondIn} c_1 \leftharpoonup_{isOperationOf} p) \vee$
$\qquad\quad (c \rightharpoonup_{isSecondIn} a \leftharpoonup_{isFirstIn} c_1 \leftharpoonup_{isOperationOf} p))\ ;$

OM13 : $\forall\, pr : V_{Propagation};\ p : V_{Operation};\ c : V_{Class} \mid$
$\quad p \rightharpoonup_{isPropagated} pr \leftharpoonup_{isTargetOf} c$
$\qquad \bullet\ (\exists\, p_1 : V_{Operation} \bullet p_1 \rightharpoonup_{isOperationOf} c \wedge p.name = p_1.name)\ ;$

OM14 : $\forall\, c : V_{Class} \mid c.isDerived = TRUE$
$\quad \bullet \forall\, a : V_{Attributes} \mid c \leftharpoonup_{isAttributeOf} a \bullet a.isDerived = TRUE\ ;$

DM1 : $\forall\, sd_1, sd_2 : V_{StateDiagram} \mid sd_1 \neq sd_2 \bullet sd_1.name \neq sd_2.name\ ;$

DM2 : $\forall\, s_1, s_2 : V_{State} \mid s_1 \neq s_2 \bullet s_1.name \neq s_2.name\ ;$

DM3 : $\forall\, e_1, e_2 : V_{Event} \mid e_1.name = e_2.name$
$\quad \bullet\ e_1 = e_2 \vee \neg\, (e_1 \rightharpoonup_{causes} \rightharpoonup_{comesFrom} \leftharpoonup_{comesFrom} \leftharpoonup_{causes}\ e_2)\ ;$

DM4 : $\forall\, s : V_{State} \mid s.flag = final \bullet degree(\leftharpoonup_{comesFrom}, s) = 0\ ;$

DM5 : $\forall\, t : V_{Transaction} \mid degree(\leftharpoonup_{receivesEventFrom}, t) = 1$
$\quad \bullet\ degree(\leftharpoonup_{isSentOf}, t) = 1\ ;$

DM6 : $\forall\, su : V_{Superstate}$
$\quad \bullet\ \#\{s : V_{State} \mid su \rightharpoonup_{contains} s \wedge s.flag = initial\} \leq 1\ ;$

DM7 : $\forall\, su : V_{Superstate} \mid degree(\leftharpoonup_{goesTo}, su) > 0$
$\quad \bullet\ (\exists_1\, s : V_{State} \bullet su \rightharpoonup_{contains} s \wedge s.flag = initial\ )\ ;$

DM8 : $isForest(eGraph(\rightharpoonup_{contains}))\ ;$

DM9 : $\forall\, std : V_{StateDiagram}$
$\quad \bullet\ (\exists_1\, s : V_{State} \bullet s \rightharpoonup_{isUsedIn} std \wedge \mathrm{degree}(\leftharpoonup_{contains}, s) = 0 \wedge$
$\qquad s.flag = initial)\ ;$

DM10 : $\forall\, s : V_{State};\ su : V_{Superstate};\ csu : V_{ConcurrentSuperstate} \mid$
$\quad csu \rightharpoonup_{contains} su(\rightharpoonup^{)}_{contains} +s$
$\qquad \bullet\ (\forall\, s_1 : V_{State} \mid s_1 \leftharpoonup_{goesTo} \rightharpoonup_{comesFrom}\ s \vee$
$\qquad\qquad s_1 \leftharpoonup_{comesFrom} \rightharpoonup_{goesTo}\ s$
$\qquad\quad \bullet\ su(\rightharpoonup_{contains})^{+} s_1\ )\ ;$

DM11 : $\forall\, s : V_{State} \mid s.flag = splitting \vee s.flag = synchronization$
$\quad \bullet\ degree(\rightharpoonup_{contains}, s) = 0\ ;$

DM12 : $\forall\, s : V_{State} \mid s.flag = splitting$
$\quad \bullet\ (\forall\, s_1 : V_{State} \mid s \leftharpoonup_{comesFrom} \rightharpoonup_{goesTo}\ s_1$
$\qquad \bullet\ (\exists_1\, csu : V_{ConcurrentSuperstate} \bullet csu(\rightharpoonup_{contains})^{+} s_1))\ ;$

DM13 : $\forall\, s : V_{State} \mid s.flag = synchronization$
$\quad \bullet\ (\forall\, s_1 : V_{State} \mid s_1 \leftharpoonup_{comesFrom} \rightharpoonup_{goesTo}\ s$
$\qquad \bullet\ (\exists_1\, csu : V_{ConcurrentSuperstate} \bullet csu(\rightharpoonup_{contains})^{+} s_1))\ ;$

DM14 : $\forall\, e : V_{Event};\ o : V_{Operation} \mid e \rightharpoonup_{mapsTo} o$
$\bullet\ e \rightharpoonup_{isUsedIn} \rightharpoonup_{refersTo} \leftharpoonup_{isOperationOf} o\ ;$

DM15 : $\forall\, a : V_{Action} \bullet\ degree(\rightharpoonup_{isMappedTo}, a) + degree(\rightharpoonup_{triggers}, a) = 1\ ;$

FM1 : $\forall\, dfd_1, dfd_2 : V_{DataFlowDiagram} \mid dfd_1 \neq dfd_2$
$\bullet\ dfd_1.name \neq dfd_2.name\ ;$

FM2 : $\forall\, cf : V_{CreationFlow} \bullet\ cf \rightharpoonup_{endsAt} \subseteq V_{DataStore}\ ;$

FM3 : $\forall\, v, w : V_{RealNodeElement};\ d : V_{DataFlow} \mid v \leftharpoonup_{startsAt} d \rightharpoonup_{endsAt} w$
$\bullet\ type(v) = Process \lor\ type(w) = Process\ ;$

FM4 : $\forall\, c : V_{Controlflow} \bullet\ c \rightharpoonup_{startsAt} \cup\ c \rightharpoonup_{endsAt} \subseteq V_{Process}\ ;$

FM5 : $\forall fc : V_{FlowConnector} \mid degree(\leftharpoonup_{startsAt}, fc) = 1$
$\bullet\ degree(\leftharpoonup_{endsAt}, fc) \geq 2\ ;$

FM6 : $\forall\, v, w : V_{NodeElement};\ d_1;\ d_2 : V_{DataFlow};\ fc : V_{FlowConnector} \mid$
$v \leftharpoonup_{startsAt} d_1 \rightharpoonup_{endsAt} fc \leftharpoonup_{startsAt} d_2 \rightharpoonup_{endsAt} w$
$\bullet\ type(v) = Process \lor\ type(w) = Process\ ;$

FM7 : $\forall\, dp : V_{Duplicator} \bullet\ degree(\leftharpoonup_{startsAt}, dp) = 0\ ;$

FM8 : $\forall\, v : V_{NodeElement};\ w : V_{RealNodeElement};\ d : V_{DataFlow};\ dp : V_{Duplicator} \mid$
$v \leftharpoonup_{startsAt} d \rightharpoonup_{endsAt} fc \rightharpoonup_{isDuplicatedTo} w$
$\bullet\ type(v) = Process \lor\ type(w) = Process\ ;$

FM9 : $\forall fc : V_{FlowConnector} \mid degree(\leftharpoonup_{endsAt}, fc) = 1$
$\bullet\ degree(\leftharpoonup_{startsAt}, fc) \geq 2\ ;$

FM10 : $\forall fc : V_{FlowConnector}$
$\bullet\ degree(\leftharpoonup_{startsAt}, fc) = 1 \lor degree(\leftharpoonup_{endsAt}, fc) = 1\ ;$

end.

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.