On Constraints and Queries in UML

M. Gogolla, M. Richters

Arbeitsgruppe Datenbanksysteme,
Fachbereich 3, Mathematik und Informatik,
Universität Bremen, D-28334 Bremen, Germany

Abstract: The UML has recently been extended by an Object Constraint Language (OCL). This formal language can be used for specifying constraints on a model in order to restrict possible system states. We present some examples for illustrating main concepts of OCL. Problems with the current definition of OCL resulting from imprecise or ambiguous definitions are investigated. A comparison of OCL with a language for specification of queries and integrity constraints in an Extended Entity–Relationship model shows similarities between both approaches. This comparison could lead to a better understanding of OCL.

1 Introduction

Many languages and methods for object–oriented analysis and design (OOAD) have been developed during the last decade. Although these methods share many concepts, differences in terminology and notation makes it quite difficult to directly compare or reuse design models. A promising approach in this situation is the development of the *Unified Modeling Language* (UML), which is a concerted effort to provide a single standardized language for object–oriented modeling (Rational, 1997a). It integrates and enhances concepts from three of the most popular OOAD methods, namely Booch (Booch, 1994), OMT (Rumbaugh et al., 1991), and OOSE (Jacobsen et al., 1992).

The process of standardization is organized by the Object Management Group (OMG), which issued a Request for Proposals in 1996. A first version of UML was submitted to OMG in January, 1997. The review process and feedback from a broader public audience exposed several deficiencies concerning clearness and consistency of the UML definition. In particular, it proved difficult to interpret the *UML Semantics* document (Rational, 1997b) in an unambiguous way. The given description in form of diagrams and English text was not precise enough for a complete and consistent understanding of the UML metamodel. In fact, the authors expected other individuals to express the UML semantics more precisely by using formal techniques.

The problem has been recognized and a new version (UML 1.1) with many improvements has been published recently. A major extension in this version is the introduction of an *Object Constraint Language* (OCL) (Rational, 1997c). This language is intended to facilitate specification of model properties in a formal yet comprehensible way (Warmer et al., 1997). For example, the semantics document has been completely restructured by replacing large parts of textual descriptions of constraints on the UML metamodel by equivalent OCL expressions.

Besides the aforementioned application of OCL to formalizing the UML metamodel, there is also benefit for users of UML. Since the constraint language is also a *part* of the metamodel, it can be used to express constraints on concrete system models. Previously, UML support for describing constraints was limited to annotations in form of uninterpreted textual comments. The introduction of a formal constraint language is an important step towards proper formalization of complex models. It should also facilitate the development of methods for simulation, validation and verification of system models. In the information systems field we are particularly interested in data models and their query languages. With respect to its use as a query language, the relationship of OCL to object–oriented query languages like OQL (Cattell, 1997), to (E)ER models like the original ER model (Chen, 1976), the data type based approach (dos Santos et al., 1979), INCOD (Atzeni, 1981), ECR (Elmasri et al., 1985), BIER (Eder et al., 1987), ERC (Parent and Spaccapietra, 1987), or ORAC (Wei and Teorey, 1991) should be investigated.

The structure of the paper is as follows. Section 2 introduces the concepts of OCL and Section 3 gives some examples of constraint expressions. In Section 4 we report on some problems with the current definition of OCL. Section 5 compares OCL with a language for specification of queries and integrity constraints in an Extended Entity–Relationship model, and Section 6 gives some final remarks.

2 Concepts of OCL

The main purpose of the Object Constraint Language is to specify restrictions on the possible system states with respect to a given model. An OCL expression is declarative in the sense that an expression says *what* constraint has to be maintained, not *how* this is accomplished. Therefore, specification of constraints is done on a conceptual level, where implementation aspects are irrelevant.

Evaluation of an expression always results in a value without changing the system state of a model. With respect to its evaluation strategy, the language resembles functional programming languages, and considering its use as a constraint language, there is an obvious connection to logic based languages. In fact, Sect. 5 will reveal a strong similarity to a query and constraint language for an EER model based on predicate calculus.

There are several places where OCL expressions can be used within a UML model. According to the metamodel described in Rational (1997b), any kind of ModelElement can be associated with a Constraint (ModelElement and Constraint both being classes of the UML metamodel). A Constraint has an attribute 'body' of type Expression. Any legal OCL expression may be used as a body of a constraint. For example, an attribute 'age' of a class Person may be restricted to hold only positive integer values. This may be achieved by attaching the expression `self.age >= 0` to the Person class. In general, constraints may also be related to a group of model elements, allowing to specify relationships among elements. Path expressions can be used to navigate various kinds of associations, e.g. a "recursive" children association of a class Person can be constrained to allow only unique forenames for children of a given person.

OCL expressions are not only used to define invariants on classes and other types, they also allow specification of pre– and postconditions on operations. If an operation has no side effects (i.e. the operation has the property "isQuery"), its meaning can usually be specified with an OCL expression. Examples of these can be found in the *UML Semantics* document where OCL is used extensively to define static well–formedness rules of the metamodel. Frequently needed operations are named and defined separately, thus reducing the overall length of rules.

Besides its application for improving the UML metamodel, OCL can also be utilized in concrete application models. UML users may add constraints to their models in order to restrict the possible set of system states. Because of the descriptive nature of constraints, OCL expressions can also be used for specifying queries. Since any expression is evaluated to a value of a certain type, constraints can be considered a special case where the result has to be a truth value. Queries can be very useful to validate models in the early phases of design. Tools supporting simulation or animation of models are powerful only if they provide means for querying the simulated state of a system (Richters and Gogolla, 1997).

3 OCL Examples

Let us take a closer look now at some examples of OCL expressions. The following expressions refer to Fig. 1, which shows a class model for a simple airline domain.[1]

The first example expresses an invariant on Plane instances, stating that their attribute capacity must have positive values.

(1) "The Capacity of each plane must be positive"

```
Plane
  self.capacity > 0
```

Evaluation of an expression always requires a contextual instance of a specific type. In the example, the expression `self` refers to an object of type Plane. Properties like attributes (e.g. 'capacity') may be referenced by using a dot notation. The term "property" is also used for association ends and side effect-free operations.

[1] This example has been used to illustrate some OCL features in a preliminary version of the OCL specification document, but has been dropped from the final version.

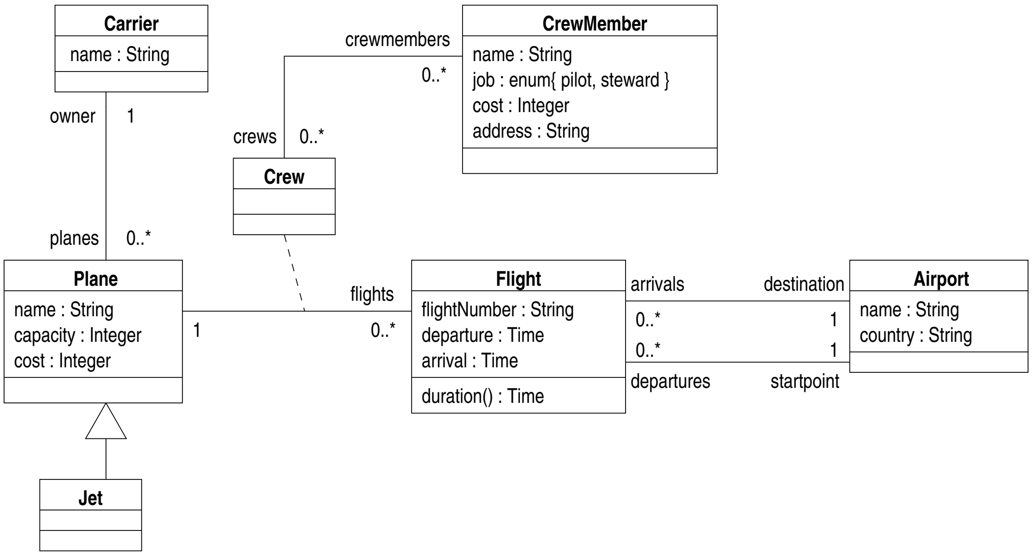

Figure 1: Class Diagram for Airline Example

To determine the names of all planes owned by a given carrier (see next example), we first have to follow the association between carrier and plane objects. The expression `self.planes` results in a set containing all elements of type Plane owned by a given carrier instance. Next, the predefined operation `collect` applies the expression `name` to every single element of this set. Generally, application of `collect` on a set results in a multiset with same cardinality but possibly duplicate values.

(2) "Return the names of all planes owned by a given carrier"

```
Carrier
  self.planes->collect(name)
```

If an expression denotes a collection (e.g. the set of planes), it is accessed by using an arrow `->` (instead of a dot) followed by a feature name. The next examples demonstrate further useful collection features. The `select` feature can be used to retrieve subsets of collection. A boolean expression given as argument is evaluated for each element of a collection, thus deciding whether an element has to be included in the result set. Similarly, a `forall` feature applies its argument expression to each element of a collection, making the whole expression true, only if the argument yields true for each element.

(3) "A carrier owns at least two planes with less than 100 seats"

```
Carrier
  self.planes->select(capacity<100)->size >= 2
```

(4) "Each plane of a carrier has a cost below 500.000"

```
Carrier
  self.planes->forall(cost<500000)
```

(5) "All planes owned by a carrier have unique names"

```
Carrier
  self.planes->forall(p1, p2 |
    p1.name = p2.name implies p1 = p2)
```

While example (2) is a query with result type Bag(String), all other expressions are constraints (i.e. their result type is boolean) that must be true in all valid system states. We consider these constraints for demonstration purposes only. They are not intended to be a complete realistic constraint set for the given class model.

4 Some Problems of OCL

In this section we investigate some aspects of OCL in more detail. We look at the type system, the concept of value and object, rules for various forms of polymorphism, rules for handling complex values and the undefined value, and some technical issues like notation and different ways of interpreting expressions. As it will turn out, there are some aspects, which are not sufficiently defined in the OCL specification to give a precise idea of their meaning.

4.1 Types

Each OCL expression has a type that belongs to one of the following groups.

- Basic types are Integer, Real, Boolean, and String.
- Collection types are Collection, Set, Sequence, and Bag.
- Special types are OclAny, OclType, and OclExpression.
- Additional types are introduced by classes, interfaces, associations and enumerations as part of a UML model.

Type hierarchy. A subtype relationship induces a hierarchy on these types (see Fig. 2). The type OclAny is the supertype of all other types. All OCL types have a set of predefined operations that can be applied to instances of a type. These operations are also called "features". For example, the type Integer provides features for arithmetic operations (addition, multiplication, etc.), and the type Set has features for common operations on sets like union, intersection, difference, etc.

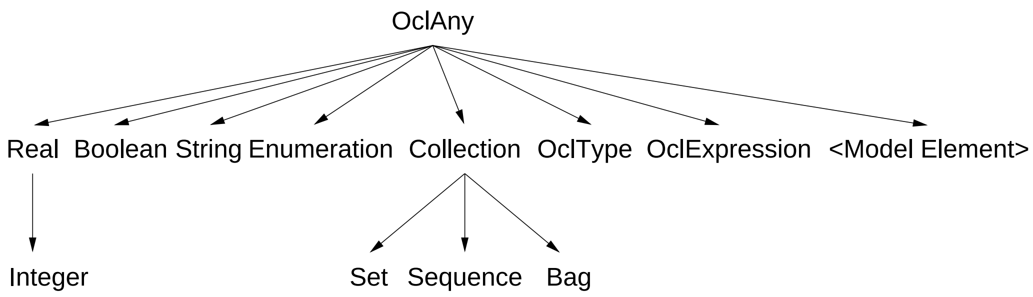

Figure 2: OCL Type Hierarchy

Features of a specific type are also available for any of its subtypes, e.g. to determine the size of a Set, Sequence, or Bag the feature 'size' defined for the Collection type can be applied. An expression is only valid, if it complies to certain rules for *type conformance*. These rules say, that an instance of type B can be used where an instance of type A is expected, if B is a (direct or indirect) subtype of A.

There are several problems with the current definition of types in OCL. First, it is not clear whether a conceptual distinction is made between data values and object instances. We consider data values to be immutable whereas object instances have a mutable state (Beeri, 1990). The terms value/object and subtype/subclass are not used consistently throughout the OCL specification document. The type hierarchy as shown in Fig. 2 has been derived from the textual description of OCL types. It is not clear whether the basic data types (Real, Integer, Boolean, String) are really (or should be) subtypes of OclAny. For example, comparison of an Integer value with a String value should be prohibited by typing rules. However, both being a subtype of OclAny, this would perfectly satisfy the type conformance rules. Consider the following feature defined for type OclAny.

```
object1 = (object2 : OclAny) : Boolean
```

Equality is a feature ("`=`") defined on values of type OclAny (`object1`) expecting as argument another value of type OclAny (`object2`). According to the type conformance rules any subtype of OclAny can be used in place. Since Integer and String are subtypes of OclAny, the expression `5 = "five"` should be type correct which certainly is not the intended behavior.

Polymorphism. The language introduces various kinds of polymorphism. We classify these according to Cardelli and Wegner, (1985).

- Inclusion polymorphism: A subtype relationship induces a partial order on OCL types.

- Parametric polymorphism: Collection types are parameterized by their element type, e.g. Set(String).

- Ad-hoc polymorphism: Overloading of operations allow their application to different argument types.

In the current language definition, there is no precise information about how these concepts are applied correctly. For example, there are no rules defining how to select a specific operation from a set of overloaded ones.

Meta types. The OCL defines special types OclAny, OclType, and OclExpression. OclAny is the supertype of all other types and provides features that are available to all subtypes. These include operations for reflective inspection of values. For example, in OCL it is possible to query the type of any value. Evaluation of the expression `3.oclType` results in `Integer` which is an instance of OclType. Applied repeatedly, the result of `Integer.oclType` is `OclType`. But what is the result of `OclType.oclType`? There is no statement about this in the OCL specification.

Expression type. The type OclExpression is a higher–order construct used for expressions that are passed as arguments to Collection operations like select, collect, forall, etc. To evaluate any of these operations, the argument expression has to be evaluated for every element of a collection. Again, OclExpression being a subtype of OclAny we can ask for the meaning of equality between values of type OclExpression. Does `2 + 2 = 1 + 3` hold, if the terms on both sides are considered to be values of OclExpression? This would presume a notion of syntactic equality.

4.2 Flattening

There are also some serious restrictions on the way types can be combined for the construction of new complex types. We have already seen that the Collection types are actually type operators, which are parameterized with a type argument. The expression Set(T) where T may be replaced by an OCL type, should yield a new set type with elements of type T. However, OCL currently does not allow usage of Collection types for T. This means that one can not create nested collections, e.g. Bag(Set(T)) is not allowed. The problem with this restriction is, that occurrences of nested values can not be completely avoided in non–trivial models. For example, the following expression determines all flights for each plane owned by a carrier (see Fig. 1).

```
Carrier
  self.planes->collect(flights)
```

The result type should be Bag(Set(Flight)). However, according to the OCL specification the result gets "automatically" flattened into a value of type Bag(Flight). The process of flattening is not described precisely. Consider the next example.

```
Sequence{ Set{1,2}, Set{3} } ⇒ Sequence{ 1,2,3 }
Sequence{ Set{2,1}, Set{3} } ⇒ ?
```

What is the result of flattening the second expression? The naive rule of removing all inner Collections and just repeating the elements in order of their appearance in the source expression leads to the different result `Sequence{ 2,1,3 }`, which is not correct, since the expressions on the left side are equal. In order to achieve a deterministic flattening semantics, and therefore equal results in the previous example, one would have to induce an order on the set elements, which is not a priori defined on all types.

4.3 Undefined Value

Another problematic field concerns the undefined value, which is introduced in order to have it, for example, as the result of an attribute application to an object instance when the attribute is not set or for incorrect operation applications like division by zero. The general OCL rule is that, if one or more parts in an expression are undefined, then the complete expression will be undefined (with two exceptions being that `true` or-ed with anything is `true` and `false` and-ed with anything is `false`). There are however cases when this strict propagation rule seems to be too strong, for instance `undefined = undefined` should yield `true` and `if true then 1 else undefined endif` should result in `1`. Also it is not clear how undefined is handled in a context where a boolean value is expected. Is `undefined` then treated as `false` or as `true`? Is it always the case that `not(undefined) = undefined` holds? How is undefined related to the universal and existential quantifier? Consider the following terms.

collection->exists(*expression*)
collection->forall(*expression*)

What happens in either case if each application of *expression* to the elements of the collection yields `undefined`?

4.4 Structured Values

For the construction of new types, the OCL provides Set, Sequence, and Bag, all of which are restricted to only one level of nesting. In general, the minimal set of type constructors in most object–oriented data models with query languages also include a tuple type, a "struct" or a similar facility for structured aggregation of values (Butterworth et al., 1991), (Deux, 1991), (Lamb et al., 1991), (Cattell, 1997). This is necessary for delivering structured or aggregated query results, e.g. for querying the departure *and* arrival times for all flights (see Fig. 1). Currently, this kind of query is not supported by OCL.

4.5 Notation

The syntactical structure of OCL constraints is mainly characterized by path expressions. Expressions are read left-to-right, where, in general, operations are applied to a value resulting from evaluation of an expression on its left side. This way, deep syntactical nesting of function application is avoided. Function definitions thus are grouped by one of their arguments – in OO terms the “receiver” of a message causing a function to be evaluated. The disadvantage of this approach is, that functions with irrelevant ordering of arguments – such as most commutative mathematical functions – have to be defined for each single argument type. For example, the union of a bag and a set is defined in different places, once for the Bag type and once for the Set type.

```
aBag.union(aSet)
aSet.union(aBag)
```

Hence, with an increasing number of arguments, feature definitions are scattered over several locations. While the previous issue may be a matter of style, some aspects of the OCL notation lead to more serious problems. Rules for a shorthand notation allow rewriting of frequently needed expressions containing `collect` terms as the following example shows.

```
Carrier
  self.planes->collect(name)  ⇒  self.planes.name
```

Again, we can find a case, where application of this shorthand notation leads to an ambiguity.

```
Set{1,2}->collect(oclType) = Bag{Int, Int}
Set{1,2}.oclType = Set(Int)
```

Naive evaluation of these expressions yields different values. The result of the first expression is of type Bag(OclType), while the second expression has type OclType. However, the shorthand rule postulates equivalence of both expressions.

The previous examples demonstrated that the Object Constraint Language has some interesting but problematic features. At first sight, these features offer flexibility and expressiveness. However, it seems that not all consequences of some design decisions have been considered in all details. A lack of orthogonality in the language makes it possible to construct examples where the semantics of evaluation is not given or is questionable.

5 Comparison of OCL with an EER approach

As a preparation for the comparison of OCL and an EER calculus (Gogolla and Hohenstein, 1991) we shortly demonstrate how the presented UML example of an airline domain (Fig. 1) looks like in the EER approach. One equivalent EER schema (other solutions are possible) is pictured in Fig. 3. Attributes are omitted in the diagram for brevity.

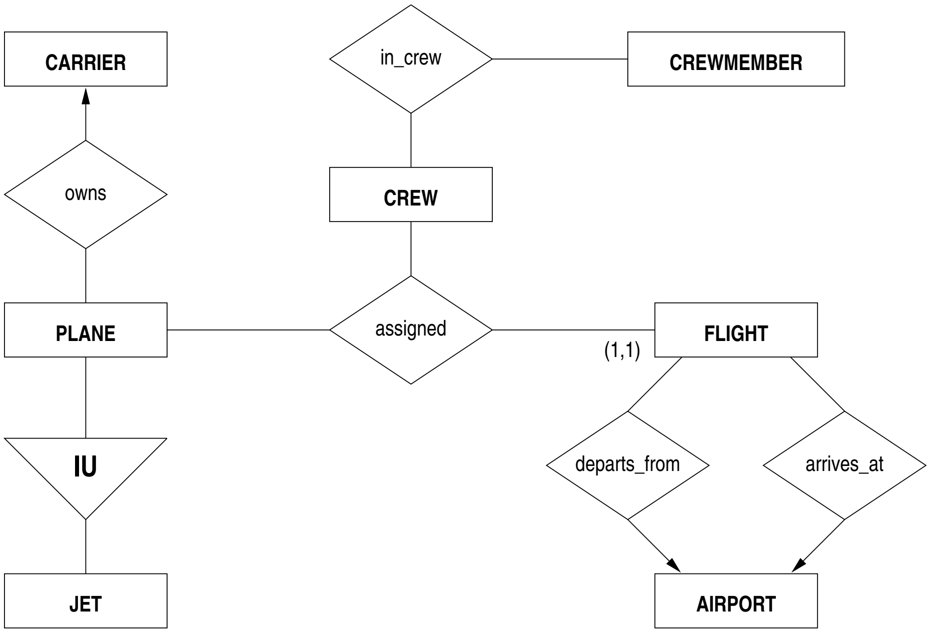

Figure 3: EER Diagram for Airline Example

In the following EER formulations of the five OCL expressions from Sect. 3 we make use of one derived schema component serving as a shorthand for the set-valued rule-navigating UML expressions. From the relationship 'owns' we derive a set-valued attribute

planes : CARRIER $\rightarrow$ set(PLANE)

by defining

planes(c:CARRIER) := BTS -[p | (p:PLANE) $\wedge$ owns(c,p)]-.

Thus the derived attribute 'planes' represents the relationship 'owns' from the viewpoint of a carrier (analogously we could do for the other side of the relationship, but this is not needed for the following examples). The definition of the derived attribute 'planes' involves a query term retrieving all planes connected via the relationship 'owns' to a given carrier. As in OCL, generally the result of a query is a multiset which is converted in the given definition to a set by the function 'BTS' which stands for 'Bag To Set'.

(1) "The Capacity of each plane must be positive"

$\forall$ (p:PLANE) capacity $>$ 0

(2) "Return the names of all planes owned by a given carrier"

-[name(p) | (p:planes(c))]-

(3) "A carrier owns at least two planes with less than 100 seats"

$\forall$ (c:CARRIER)
CNT -[p | (p:planes(c)) $\wedge$ capacity(p)<100]- $\geq$ 2

(4) "Each plane of a carrier has a cost below 500.000"

$\forall$ (p:planes(c)) cost(p) $<$ 500000

(5) "All planes owned by a carrier have unique names"

$\forall$ (c:CARRIER) (p1,p2:planes(c))
name(p1)=name(p2) $\Rightarrow$ p1=p2

The main difference between the OCL formulations and the EER calculus can be summarized as follows ($\delta = v : s$ is a declaration binding a variable v to a set s, φ is a formula, and τ is a result term).

1. Query expressions frequently are given by explicit SQL-like terms: -[τ | $\delta \wedge \varphi$]- is a short notation for SELECT τ FROM δ WHERE φ
2. Quantifiers are used in a more traditional way in the form $\forall/\exists(\delta)\varphi$.

Let us now compare both OCL and EER calculus by means of some selected topics.

Purpose: Both the EER calculus and OCL are intended to specify declarative constraints in order to restrict the possible system states to desired ones. Both also allow to query the current state of a system in the same language framework.

Semantics: The EER calculus has a completely worked out formal semantics based on set theory, and equivalence rules for EER calculus expressions are known. OCL deserves yet a theoretic foundation.

Safeness: The EER calculus is proved to be safe in the sense that all expressions and therefore in particular all query expressions yield a finite result. In the present version of OCL this is not true, for example, `Integer.allInstances` returns the infinite set of all integer values.

Computational power: Due to its structuring mechanisms the EER calculus is able to represent the result of a query in various forms. In contrast to this, OCL flattens collections of collections automatically. Flattening can also be achieved in the EER calculus but this is done by explicit formulations.

Notation: OCL directly allows to navigate through a class model in a path expression-like style. This enhances in certain cases the readability of expressions in comparison to the respective logic-based EER calculus formulations with its SQL-like query notation. However, by employing derived attributes, especially for relationships, EER calculus expressions become more user-friendly.

Orthogonality: The EER calculus is completely orthogonal, i.e. a concept can be applied if the syntactical context fits. In contrast to this, certain OCL language elements need to be restricted to special situations in order to be applied meaningfully, e.g. `allInstances` should be applied only to finitely interpreted object classes and finitely interpreted data types and `oclType` should be applied only to direct model elements.

6 Conclusion

OCL offers with respect to UML modeling advantages both in giving a better understanding of UML itself (because OCL is used for making precise the UML semantics) and in giving developers the possibility to sharpen their models. However, the present concepts and their semantics of OCL could be improved because its interpretation is partly incomplete and in some cases inconsistent.

On the other hand there are similar well-developed approaches like the EER calculus which have a precise mathematical semantics but which are with respect to notation a bit more elaborate and perhaps not so close to programming languages notation as OCL. The concepts of OCL however improve the UML modeling facilities and can be employed in a successful manner for various purposes. Most of the OCL semantics could be explained in a way analogously to, for instance, the way the EER calculus semantics is defined.

References

ATZENI, P., BATINI, C., LENZERINI, M., and VILLANELLI, F. (1981): INCOD: A System for Conceptual Design of Data and Transactions in the Entity-Relationship Model. In P. P. Chen (ed.): *Proc. 2nd Int. Conf. on the Entity-Relationship Approach (ER'81).* North Holland, 375–410.

BEERI, C. (1990): A Formal Approach to Object-Oriented Databases. *Data & Knowledge Engineering, 5(4), 353–382.*

BOOCH, G. (1994): Object–Oriented Analysis and Design with Applications. Benjamin/Cummings.

BUTTERWORTH, P., OTIS, A., and STEIN, J. (1991): The GemStone Object Database Management System. *Communications of the ACM, 34(10), 64–77.*

CARDELLI, L., and WEGNER, P. (1985): On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys, 17(4), 471–522.*

CATTELL, R. G. G. (1997): The Object Database Standard: ODMG 2.0. Morgan Kaufmann Publishers, Inc.

CHEN, P. P. (1976): The Entity-Relationship Model – Toward a Unified View of Data. *ACM Trans. on Database Systems, 1(1), 9–36.*

DEUX, O. (1991): The O2 System. *Communications of the ACM, 34(10), 34–48.*

DOS SANTOS, C. S., NEUHOLD, E. J., and FURTADO, A. L. (1979): A Data Type Approach to the Entity-Relationship Approach. In P. P. Chen (ed.): *Entity-Relationship Approach to Systems Analysis and Design. Proc. 1st International Conference on the Entity-Relationship Approach. Los Angeles, California, USA.* North Holland, 103–119.

EDER, J., KAPPEL, G., TJOA, A. M., and WAGNER, R. R. (1987): BIER — The Behavior Integrated Entity-Relationship Approach. In S. Spaccapietra (ed.): *Entity-Relationship Approach: Ten Years of Experience in Information Modeling, Proceedings of the Fifth International Conference on Entity-Relationship Approach, Dijon, France, November 17-19, 1986.* North-Holland, 147–166.

ELMASRI, R., WEELDREYER, J., and HEVNER, A. (1985): The Category Concept: An Extension to the Entity Relationship Model. *Data & Knowledge Engineering, 1, 75–116.*

GOGOLLA, M. and HOHENSTEIN, U. (1991): Towards a Semantic View of an Extended Entity-Relationship Model. *ACM Trans. on Database Systems, 16(3), 369–416.*

JACOBSEN, I., CHRISTERSON, M., JONSSON, P., and OVERGAARD, G. (1992): Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley.

LAMB, C., LANDIS, G., ORENSTEIN, J., and WEINREIB, D. (1991): The ObjectStore Database System. *Communications of the ACM, 34(10), 50–63.*

PARENT, C. and SPACCAPIETRA, S. (1989): Complex Objects Modeling: An Entity-Relationship-Approach. In: S. Abiteboul, P. C. Fischer, and H.-J. Schek, (eds.): *Nested Relations and Complex Objects, Papers from the Workshop "Theory and Applications of Nested Relations and Complex Objects", Darmstadt, Germany, April 6-8, 1987*, vol. 361 of *Lecture Notes in Computer Science*, 272–296.

RATIONAL (1997a): Unified Modeling Language, Version 1.1. Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951 (USA), URL: `http://www.rational.com`.

RATIONAL (1997b): UML Semantics. Published as part of RATIONAL (1997a).

RATIONAL (1997c): Object Constraint Language Specification. Published as part of RATIONAL (1997a).

RICHTERS, M. and GOGOLLA, M. (1997): A Web-based Animator for Validating Object Specifications. In: B. C. Desai and B. Eaglestone (eds.): *IDEAS'97: International Database Engineering & Applications Symposium, Montreal, Canada, August 25–27, 1997*, 211–219.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., and LORENSON, W. (1991): Object–Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs (NJ).

WARMER, J., HOGG, J., COOK, S., and SELIC, B. (1997): Experience with Formal Specification of CMM and UML. In: H. Kilov and B. Rumpe, (eds.): *Proceedings of the ECOOP'97 Workshop on Precise Semantics for Object–Oriented Modeling Techniques, Jyväskylä, Finland, 10 June 1997.* TU München, TUM-I9725.

WEI, G. and TEOREY, T. (1991): The ORAC Model: A Unified View of Data Abstraction. In: T. J. Teorey, (ed.), *Proc. 10th Int. Conf. on Entity-Relationship Approach (ER'91). ER Institute, Pittsburgh (CA)*, 31–58.