Proc. 17th Int. Conf. Conceptual Modeling (ER'98), Tok-Wang Ling (ed.), Springer LNCS, 1998

On Formalizing the UML Object Constraint Language OCL

Mark Richters and Martin Gogolla

University of Bremen, FB 3, Computer Science Department
Postfach 330440, D-28334 Bremen, Germany
{mr|gogolla}@informatik.uni-bremen.de,
WWW home page: http://www.db.informatik.uni-bremen.de

Abstract. We present a formal semantics for the Object Constraint Language (OCL) which is part of the Unified Modeling Language (UML) – an emerging standard language and notation for object-oriented analysis and design. In context of information systems modeling, UML class diagrams can be utilized for describing the overall structure, whereas additional integrity constraints and queries are specified with OCL expressions. By using OCL, constraints and queries can be specified in a formal yet comprehensible way. However, the OCL itself is currently defined only in a semi-formal way. Thus the semantics of constraints is in general not precisely defined. Our approach gives precise meaning to OCL concepts and to some central aspects of UML class models. A formal semantics facilitates verification, validation and simulation of models and helps to improve the quality of models and software designs.

1 Introduction

Methods for object-oriented analysis and design have brought a great number of languages and notations supporting many aspects of the software specification process. The *Unified Modeling Language* (UML) is the result of an effort in developing a single standardized language for object-oriented modeling [22]. It integrates and enhances concepts from many analysis and design methods, including Booch [4], OMT [25], and OOSE [16].

Part of the latest UML proposal is the *Object Constraint Language* (OCL) [24]. This language is intended to facilitate specification of model properties in a formal yet comprehensible way [28]. Previously, UML support for describing constraints was limited to annotations in form of uninterpreted textual comments. The introduction of a formal constraint language therefore is an important step towards proper formalization of complex models.

However, the OCL itself is mainly defined in a semi-formal way by using English text descriptions, a context-free grammar specifying the concrete syntax of OCL and many examples illustrating the intended meaning of expressions. While this style of presentation is perfectly well-suited to introduce and demonstrate the concepts of OCL, we argue that a thorough understanding of OCL semantics requires a formal definition. The authors themselves emphasize that currently no complete formal semantics exists [18]. Due to this semi-formal nature of the OCL definition some issues have already been identified that may lead to problems or open questions [14, 15].

As a simple example, we present an OCL expression where the result cannot be unambiguously determined. Consider a query which builds a list containing the names of persons working at a car rental station. We could use the following OCL expression for this purpose.

```
     rs :RentalStation
(1)    rs.employee->iterate(p :Person;
          names :String = "" | names.concat(p.lastname))
```

The expression iterates over the elements in a set of persons (`rs.employee`) and adds the last name of each person in this set to an initially empty string. The result will be a string containing all last names of a rental station's employees. One problem here is that there is no statement in OCL about the order of evaluation, hence evaluations may yield different results caused by different iteration sequences. This shows that important aspects of the iterate expression are underspecified, even worse, since most operations on collections are defined in terms of iterate expressions their behavior is also not precisely defined. In our paper we will present a solution to this problem by defining a deterministic evaluation semantics and by identifying necessary preconditions for application of iterate expressions.

We present a formalization of OCL and certain aspects of UML class models. To this end, we define syntax and semantics of OCL expressions and give examples for their evaluation. It is our goal to keep as close as possible with the intended semantics of OCL. However, where the intended semantics is not obvious or appears problematic we do not refrain from suggesting an alternative approach.

In our view a formalization of the OCL is beneficial for achieving the following goals: (1) Improvement of the OCL itself by pointing out some potential problems with its current definition, (2) a more precise understanding of UML class models and their interpretation, (3) a solid foundation for implementing CASE tools supporting simulation and validation of UML models, and (4) integration of UML with classical approaches for conceptual modeling of information systems.

Our work is related to recent approaches handling formal aspects of UML and other object-oriented methods. Work has been done on the basis of well-established traditional approaches to specification like Z and VDM: [26, 11, 12, 10] focuses on the UML type system and class models, a general integration of Z with object-orientation is discussed in [9], and in [19, 3] an object calculus enhancing the expressibility of object-oriented notations has been proposed. Other approaches treat in detail the UML predecessor OMT [2, 27], and in particular class diagrams [5] in connection with Larch are discussed. [6] sketches a general scenario for most UML diagrams without going into technical details, and [21] presents a general framework for relationships and Use Cases in UML.

In the information systems field we are particularly interested in data models and their query languages. We see a strong resemblance between the UML language and notation for class diagrams and traditional Entity-Relationship approaches, e.g. [8, 13]. An example comparing a UML class diagram with an EER diagram and integrity constraints in OCL with constraints specified in the EER calculus can be found in [14]. The UML might also be appropriate for modeling object-oriented database systems according to the ODMG standard [7]. In fact, OCL seems to be not so far away from the ODMG query language OQL.

The rest of the paper is organized as follows. In Sect. 2 we briefly introduce the OCL by examples and illustrate their application to UML class models for specifying constraints and queries. We start our formal framework by first developing a formal description of basic and complex types in Sect. 3. In Sect. 4 we propose a model to formalize structural properties of class models which may be referenced in OCL expressions. Syntax and semantics of expressions is then defined in Sect. 5. We close with a summary and point out problems that still need some further consideration.

2 UML and the Object Constraint Language

The primary purpose of OCL is to augment a model with additional information that often (if at all) cannot be expressed appropriately in UML. This information is given by constraints which in general are easier to specify in a textual notation than in a graphic-oriented language. The OCL provides a flexible and concise notation for specifying a broad range of constraints. For example, a simple constraint might just restrict attribute values to some subset of an otherwise fixed domain (e.g., stating that the age of a person must be a positive integer), or it might describe some complex relationship between several objects.

The intended effect of a constraint specification is a restriction on possible system states and transitions 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 mainly irrelevant. We only give some basic examples of constraints to illustrate the general structure of OCL expressions. Refer to [24] for a complete overview of OCL.

All examples in the following sections refer to the class diagram shown in Fig. 1. The diagram is a simplified model of a car rental company having several rental stations each employing a number of persons. A company has a fleet of cars and motorcycles which are generalized into the class vehicle. We do not claim this model to be adequate for representing the real world, rather our intention here is to provide a simple example using the UML concepts of class, attribute, operation, association, and generalization.

Evaluation of an expression always requires a contextual instance of a specific type. In the example (2) given below, the expression `self` refers to an object of type *Person*. Class properties like attributes (e.g., *age*) may be referenced by using a dot notation. Thus expressions generally expand from left to right.

(2)
```
Person
  self.age > 0
```

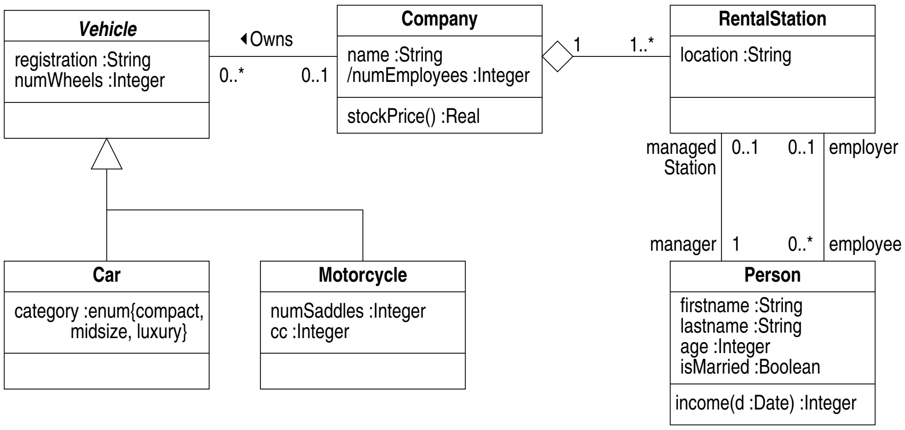

Fig. 1. UML class diagram

The term "property" is also used for referring to association ends (i.e., the role an object plays in an association, e.g., a person is an *employee* in a rental station) and side effect-free operations (e.g., *income*). The next expression (3) ensures that all employees of a station have a minimum income at a given date. `forAll` is a predefined OCL operation testing whether a predicate holds for each member of a collection (a set of employees in this case).

(3)
```
RentalStation
  self.employee->forAll(p | p.income("98/03/01") > 2000)
```

The OCL is also suitable for specifying queries. In fact, constraints can be considered a special kind of queries which are required to have a boolean result type whereas a query may have any result type. The expression (4) selects for a given company all vehicles which have a registration number starting with the string `"R24"`. Due to subtyping rules, the result type *Set(Vehicle)* is capable of holding both car and motorcycle instances. The subexpression `vehicle` refers to the left end of the association *Owns* and provides the set of all vehicles owned by a company which is then further restricted by the select expression.

(4)
```
Company
  self.vehicle->select(v |
    v.registration.substring(1,3) = "R24")
```

3 Types

We start developing our formal definition of the Object Constraint Language by first looking at concepts for representing types and associated operations. In a way quite similar to algebraic specification, we use a signature to describe sorts and operation symbols. After defining basic sorts we proceed with sort constructors and subsort relations. Figure 2 gives an overview of types available in OCL. Arrows indicate a subtype relationship between types. Further types introduced by UML model elements will be considered in Sect. 4.

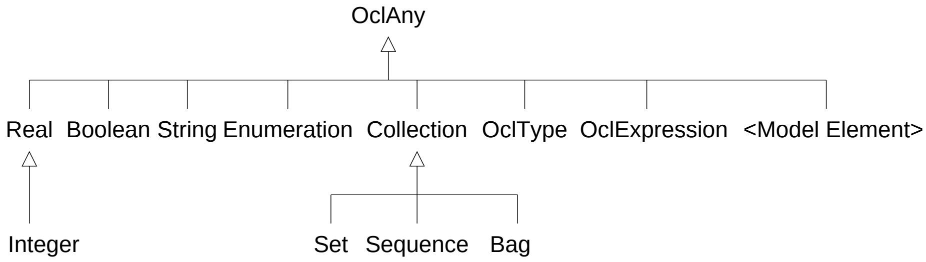

Fig. 2. OCL Type Hierarchy

3.1 Basic Types

Basic data types defined in OCL are *Boolean*, *Integer*, *Real*, and *String*. These types are independent of any particular UML model. In our approach, the syntax of basic types and their properties are described by a data signature $\Sigma_D = (S_D, \Omega_D)$ with S_D being a set of data sorts, and Ω_D a set of operations. The signature includes at least the following simple sorts as required by the OCL definition of basic types: $\{Boolean, Integer, Real, String\} \subseteq S_D$.

Interpretation of basic sorts is as usual but we extend each set with a special value $\perp$ denoting the undefined value. A sort $s \in S_D$ is mapped to a semantic domain $\mathcal{D}_s$ by a function $I : s \mapsto \mathcal{D}_s$ as follows (A^* is the set of all sequences over a finite alphabet A): $I(Integer) = \mathbb{Z} \cup \{\perp\}$, $I(Real) = \mathbb{R} \cup \{\perp\}$, $I(Boolean) = \{\mathrm{true}, \mathrm{false}\} \cup \{\perp\}$, $I(String) = A^* \cup \{\perp\}$.

Enumeration types used in a UML model extend the set of basic types. Their domain is given by an enumeration of literal values. For example, the enumeration `enum{compact,midsize,luxury}` in Fig. 1 can be assigned a type *CategoryKind* which is interpreted as $I(CategoryKind) = \{\mathrm{compact}, \mathrm{midsize}, \mathrm{luxury}\} \cup \{\perp\}$.

A set of operations is defined on basic sorts. Note that some operations have the same (overloaded) name symbol and can only be distinguished by also looking at their argument sorts.

$$
\begin{aligned}
&\{\vee, \wedge : Boolean \times Boolean \to Boolean, \neg : Boolean \to Boolean, \dots\} \cup \\
&\{+, -, *, /, \mathrm{div}, \mathrm{mod}, \max : Integer \times Integer \to Integer, \dots\} \cup \\
&\{<, \leq, \geq, >: Integer \times Integer \to Boolean, \dots\} \cup \\
&\{+, -, *, /, \max, \min : Real \times Real \to Real, \mathrm{floor} : Real \to Integer, \dots\} \cup \\
&\{<, \leq, \geq, >: Real \times Real \to Boolean, \dots\} \cup \\
&\{\mathrm{concat} : String \times String \to String, \mathrm{size} : String \to Integer, \dots\} \subseteq \Omega_D.
\end{aligned}
$$

We only give two examples for interpretations of operations. In OCL, care has to be taken when passing undefined values as arguments. In general, all operations have a strict evaluation semantics. If at least one argument is undefined, then the whole operation yields the undefined value as a result. Exceptions of this rule are the boolean operations $\wedge$ and $\vee$. False and-ed with anything yields false, and true or-ed with anything yields true.

$$I(+)(i_1, i_2) = \begin{cases} i_1 + i_2, & \text{if } i_1 \neq \bot \text{ and } i_2 \neq \bot \\ \bot, & \text{otherwise} \end{cases}$$

$$I(\vee)(b_1, b_2) = \begin{cases} \text{true}, & \text{if } b_1 = \text{true or } b_2 = \text{true} \\ \text{false}, & \text{if } b_1 = \text{false and } b_2 = \text{false} \\ \bot, & \text{otherwise} \end{cases}$$

The OCL rules for undefined values lead to a rather unexpected behavior of `exists` and `forAll` expressions. If one element of a collection is undefined then the result of the whole expression will be undefined. However in case of `forAll` we would expect a result of false, and in case of `exists`, there well might be some other element making the whole expression true. Thus a more natural behavior could be achieved by requiring all operations with boolean result to yield false in case of undefined arguments.

3.2 Complex Types

In OCL multi-valued expressions are described by collection types. Sets, bags, and lists can be constructed by providing a type parameter for their element type. For example, the type *Sequence(Integer)* denotes list values with integer elements. However, collection types may not be nested, therefore the explicit construction of more complex types is not possible in OCL. When an evaluation yields a value with deeper nesting, a flattening process "automatically" transforms this value to a set, bag or sequence containing only basic data values. We lift this restriction for the sake of orthogonality by allowing an arbitrary number of constructor applications. Complex structures can not be avoided in non-trivial models. For example, the expression `c.rentalStation->collect(employee)` with `c` being an object of class *Company* selects all rental stations of a company together with their employees. The result type should be *Bag(Set(Person))*. However, according to the OCL definition the result gets automatically flattened into a value of type *Bag(Person)*. How this flattening works is described only by means of the following example [24]:

```
Set{ Set{1,2}, Set{3,4}, Set{5,6} } ⇒ Set{ 1, 2, 3, 4, 5, 6 }
```

The naive rule of removing all inner collections and just repeating the elements in order of their appearance in the source expression seems to suffice in this example, but we can easily construct a case where it does not produce a well-defined result. Consider the expression `Sequence{Set{p1,p2}}` with `p1,p2` being variables holding objects of type *Person*. Is the result of flattening `Sequence{p1,p2}` or `Sequence{p2,p1}`? The problem here is that mapping a set into a sequence requires an order on the set elements. For achieving 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. The requirement of generally having an order on all types seems rather strong. We therefore suggest to keep the structure of evaluation results. Where flattening of nested structures is desirable, explicit transformations have to be defined and used. Thus, the following definition generalizes the idea of nested data structures by the notion of sort expressions that can be recursively applied to construct complex structures.

Definition 1 (Syntax of Sort Expressions) Let S be a set of sort symbols. The set of sort expressions $S_{\mathrm{Expr}}(S)$ is defined as follows:

i. If $s \in S$ then $s \in S_{\mathrm{Expr}}(S)$.
ii. If $s \in S_{\mathrm{Expr}}(S)$ then $\mathit{Set(s)}, \mathit{Sequence(s)}, \mathit{Bag(s)} \in S_{\mathrm{Expr}}(S)$.
iii. If $s \in S_{\mathrm{Expr}}(S)$ then $\mathit{Collection(s)} \in S_{\mathrm{Expr}}(S)$.
iv. $\mathit{Any} \in S_{\mathrm{Expr}}(S)$.

The initial set of sort expressions (i) is given by the set of basic sorts. For now, we consider only the data sorts in S_D as defined in the previous section. Later, in Sect. 4, we will extend this set by classifier types introduced by UML class models. Complex structures can be modeled by using sort constructors *Set*, *Sequence*, and *Bag* (ii). A *Collection* type (iii) is not strictly necessary since it is defined in OCL as an abstract super type of *Set*, *Sequence*, and *Bag*. Thus from a semantic point of view, there will be no values having the most specific type *Collection*. However, it is convenient to define operations common to type *Set*, *Sequence*, and *Bag* only once by means of an abstract super type. *OclAny* (or just *Any*) is the top of the type hierarchy in OCL (iv). All other types are either direct or indirect subtypes of *Any*.

Sort expressions are interpreted as sets of possible values. In the following definition, $\mathcal{F}(I(s))$ denotes the set of all finite subsets of $I(s)$, and $\mathcal{B}(I(s))$ is the set of all finite multisets over $I(s)$.

Definition 2 (Semantics of Sort Expressions) The semantics of sort expressions $s \in S_{\mathrm{Expr}}(S)$ is defined as follows:

i. $I(s)$ is defined in Sect. 3.1.
ii. $I(\mathit{Set}(s)) = \mathcal{F}(I(s)) \cup \{\bot\}$,
$I(\mathit{Sequence}(s)) = (I(s))^* \cup \{\bot\}$,
$I(\mathit{Bag}(s)) = \mathcal{B}(I(s)) \cup \{\bot\}$.
iii. $I(\mathit{Collection}(s)) = I(\mathit{Set}(s)) \cup I(\mathit{Sequence}(s)) \cup I(\mathit{Bag}(s))$.
iv. $I(\mathit{Any}) = \bigcup_{s \in S_{\mathrm{Expr}}(S) \setminus \{\mathit{Any}\}} I(s)$.

A set of operations $\Omega_{S_{\mathrm{Expr}(S)}}$ is defined on sort expressions. These operations can be used to express functions (also called features in OCL) over collection types. Some examples are listed in the following table.

OCL Feature	Operation
`set->union(set2 :Set(T)) :Set(T)`	union : $Set(s) \times Set(s) \to Set(s)$
`bag->including(object :T) :Bag(T)`	including : $Bag(s) \times s \to Bag(s)$
`collection->size :Integer`	size : $Set(s) \to Integer$
	size : $Sequence(s) \to Integer$
	size : $Bag(s) \to Integer$

The left column displays signatures of features as they are defined in [24]. The left-hand side of an arrow[1] denotes an instance of the indicated type. A corresponding operation in our language is given in the right column for each case. Inherited features on type *Collection* (e.g., `size`) require specialized operations for each subtype. The appropriate operation will be selected by determining whether the collection expression results either in a set, bag, or sequence value.

A further group of operations serves as constructor functions for representing literals in OCL. For example, a literal `Set{2,3}` will be expressed as an operation $\mathrm{MkSet}(2, 3)$. Features like `exists`, `forAll`, etc. are operations with an argument of type *OclExpression*. We do not support this type since it mixes different levels of abstraction [14]. Instead we define these operations as part of the expression syntax in Sect. 5.

In this section, we introduced sort expressions with associated operations. By combining them with our initial data signature we obtain an extended signature $\Sigma = (S_{\mathrm{Expr}}(S_D), \Omega_{S_{\mathrm{Expr}}(S_D)} \cup \Omega_D)$.

4 UML Class Models

The relationship between the UML metamodel and OCL is informally explained in [24]. Since components of a UML model form the basic building blocks for constraint expressions, we emphasize the need for a precise interface. The following features of a UML class model are accessible in OCL expressions:

- *Classifiers:* Each Classifier[2] introduces a distinct type (e.g., *Person*, *Company*, etc.). Class types are probably the most frequently used types in OCL expressions. Further kinds of Classifiers in the core model are Datatypes and Interfaces. Since basic data types are already included in our data signature we do not consider them here again. Furthermore the subtype hierarchy induced on classifiers by generalization/specialization is utilized to employ a kind of inclusion polymorphism in OCL.
- *Attributes:* Structural properties of classifiers are described by attributes (e.g., *name*, *location*, *isMarried*, etc.).
- *Associations:* An association connects two or more classifiers (not necessarily being distinct). Complex expressions referring to several class types can be constructed by navigation along these associations (e.g., "retrieve

[1] An arrow indicates a collection expression whereas a dot generally is used for single-valued expressions.
[2] Terms set in a serif-less font refer to elements of the UML metamodel (see [23]).

all persons working as employees for a company"). For navigation purposes, the association ends are uniquely identified by role names. Aggregational and compositional relationships can be treated uniformly like associations in OCL.
- *Operations:* Operations and methods that are marked as being side effect-free (i.e., they have the property *isQuery*) may be used in an OCL expression (e.g., `p.income(date)` returns the income of a person `p` at a specific date).
- *Enumerations:* Enumeration types extend the set of basic types (see Sect. 3.1).

With regard to the formalization of OCL it is sufficient to consider only the previously listed items. These properties all describe static aspects of a UML model that do not change over time. For the specification of temporal aspects OCL provides pre- and postconditions. Since these are primarily used to specify effects of method definitions, we do not consider them here.

Let $\mathcal{A}_{\text{attr}}$ be a set of attribute names and $\mathcal{A}_{\text{role}}$ a set of role names. We define a UML class model as a structure $\mathcal{M} = (C, \prec, \textsc{Att}_c, \textsc{Op}_c, \textsc{Assoc}_c)$ with the following meaning:

- C is a set of classifiers.
- $\prec$ is a partial order on C reflecting the generalization hierarchy of classifiers.
- $\textsc{Att}_c$ is a set of operation signatures $a_c : c \mapsto s$ for functions mapping an object of type c to an associated attribute value of sort s. The attribute symbol a_c is an element of $\mathcal{A}_{\text{attr}}$.
- $\textsc{Op}_c$ is a set of signatures $\omega_c : c \times s_1 \ldots s_n \mapsto s$ for side effect-free operations defined for a classifier c having result sort s. The first argument denotes the instance to which a function is applied (sometimes called the receiver of a message).
- $\textsc{Assoc}_c$ is a set of operation signatures $r_{<c,c'>} : c \mapsto s$ for functions returning instances associated with an object of type c. The operation symbol $r \in \mathcal{A}_{\text{role}}$ is chosen according to the unique role name assigned to an association end. Depending on the multiplicity of an association, the result sort s is either c' for single-valued associations, *Set(c')* for multi-valued associations (e.g., having a multiplicity of `0..*`), or *Sequence(c')* for ordered associations.

Example: A model $\mathcal{M}_{\text{Company}}$ for the class diagram in Fig. 1 can be defined as follows.

$$
\begin{aligned}
C &= \{\ \textit{Vehicle}, \textit{Car}, \textit{Motorcycle}, \textit{Company}, \textit{RentalStation}, \textit{Person}\} \\
\prec &= \{\ (\textit{Car}, \textit{Vehicle}), (\textit{Motorcycle}, \textit{Vehicle})\} \\
\textsc{Att}_c &= \{\ \text{name}_{\textit{Company}} : \textit{Company} \to \textit{String}, \\
&\qquad \text{isMarried}_{\textit{Person}} : \textit{Person} \to \textit{Boolean}, \ldots\} \\
\textsc{Op}_c &= \{\ \text{stockPrice}_{\textit{Company}} : \textit{Company} \to \textit{Real}, \\
&\qquad \text{income}_{\textit{Person}} : \textit{Person} \times \textit{Date} \to \textit{Integer}\} \\
\textsc{Assoc}_c &= \{\ \text{employee}_{<\textit{RentalStation},\textit{Person}>} : \textit{RentalStation} \to \textit{Set}(\textit{Person}), \\
&\qquad \text{manager}_{<\textit{RentalStation},\textit{Person}>} : \textit{RentalStation} \to \textit{Person}, \ldots\}
\end{aligned}
$$

With each classifier $c \in C$ we associate a sort having the same name. A possible interpretation of a classifier sort is a set of object identifiers denoting the currently existing object instances in a system state. In order to guarantee safe expressions, we postulate the set of objects in any system state being finite. We will consider the interpretation of a model in more detail in the next section.

4.1 Subsort Hierarchy

The OCL supports polymorphism by a subtype hierarchy defined on basic and complex types. This hierarchy is established by a subtype relation on the basic types *Integer* and *Real*, and a transitive subtype relation on collection types. Furthermore, it is extended by generalization hierarchies in UML models. We formalize this subtype hierarchy in our language by a subsort relation defined on sorts.

Definition 3 (Subsort Relation) Let $s, s' \in S$ and $c, c' \in C$ with $C \subset S$. A subsort relation $\leq$ is defined on S by the following rules:

i. $s \leq \mathit{Any}$.
ii. $\mathit{Integer} \leq \mathit{Real}$.
iii. If $c' \prec c$ then $c' \leq c$.
iv. If $s' \leq s$ then $\mathit{Set}(s') \leq \mathit{Set}(s)$, $\mathit{Sequence}(s') \leq \mathit{Sequence}(s)$, and $\mathit{Bag}(s') \leq \mathit{Bag}(s)$.
v. If $s' \in \{\mathit{Set}(s), \mathit{Sequence}(s), \mathit{Bag}(s)\}$ then $s' \leq \mathit{Collection}(s)$.

The semantics of the subsort relation is given by a subset condition. We require that for each two sorts $s', s \in S$ with $s' \leq s$ the domain of the specialized sort s' is a subset of the more general sort s, i.e. $I(s') \subseteq I(s)$.

4.2 System State

A UML model describes the static structure and dynamic behavior of a system. With respect to our discussion of OCL, we do not need to consider evolution of a system state over time. In general, we can concentrate on a system state given at a discrete point in time. This system state provides a context for the evaluation of OCL expressions. The set of existing object instances, their attribute values, and associations among instances constitute the state of a model. In this section, we present a simple model for objects, attributes, and navigational expressions by giving an interpretation for a model $\mathcal{M}$.

For each classifier $c \in C$ there is an infinitely large set of unique object identifiers $I(c) = \{\underline{c}_1, \underline{c}_2, \dots\} \cup \{\perp\}$. Subsorts imply a subset condition on object identifiers: $c \leq c' \Rightarrow I(c) \subseteq I(c')$. The interpretation of classifiers is pairwise disjoint if two classifiers are not related by subtyping: $\forall c, c' \in C : \neg(c \leq c' \vee c' \leq c) \Rightarrow I(c) \cap I(c') = \{\perp\}$.

We describe a model state $\sigma_{\mathcal{M}}$ at a given point t in time by a hypergraph G, i.e. $\sigma_{\mathcal{M}}(t) = G$. Nodes of the graph represent objects whereas edges represent associations among objects. An edge represents a link in UML terminology, i.e. an instantiated association. Since associations may be n-ary, edges may connect any number of nodes in the resulting graph, thus requiring the concept of hyperedges. Nodes and edges are labeled with additional information. Each object node carries information about attribute values, and each edge is labeled with information about role names.

Definition 4 (System State) Let $\mathcal{M}$ be a class model. A system state $\sigma_{\mathcal{M}}(t)$ is a labeled hypergraph $G = (V, E, \mathit{nodes}, \mathit{roles}, \mathit{ostate})$ defined as follows:

i. $V \subset \bigcup_{c \in C} I(c)$ is a finite set of nodes (object identifiers).
ii. E is a set of edges (association links) connecting nodes.
iii. $\mathit{nodes} : \begin{cases} E \to V^* \\ e \mapsto (v_1, \ldots, v_n) \end{cases}$ is a function connecting edges with nodes.
iv. $\mathit{roles} : \begin{cases} E \to \mathcal{A}^*_{\mathrm{role}} \\ e \mapsto (r_1, \ldots, r_n) \end{cases}$ is a function labeling each edge with a tuple of role names.
v. $\mathit{ostate} : \begin{cases} V \to \{f \mid f : \mathcal{A}_{\mathrm{attr}} \to \bigcup_{s \in S} I(s)\} \\ v \mapsto \{(a_1, \underline{s}_1), \ldots, (a_n, \underline{s}_n)\} \end{cases}$ is a function labeling each node with a set of attribute/value mappings.

Figure 3 shows a UML object diagram representing part of a system state for the car rental model. The graph in the diagram has nodes for a rental station object rs_1 and two persons p_1 and p_2. A set of attribute names and values is attached to each node, respectively. Edges indicate links between the station and persons. Note that edge labeling by role names enables us to distinguish the manager and employee association between rs_1 and p_1. In cases like this, where the same set of objects are linked by multiple edges, it is necessary that each edge can be unambiguously assigned to a distinct association. We identify associations with unique combinations of role names and require: $\forall e_1, e_2 : \mathit{nodes}(e_1) = \mathit{nodes}(e_2) \implies \mathit{roles}(e_1) \neq \mathit{roles}(e_2)$.

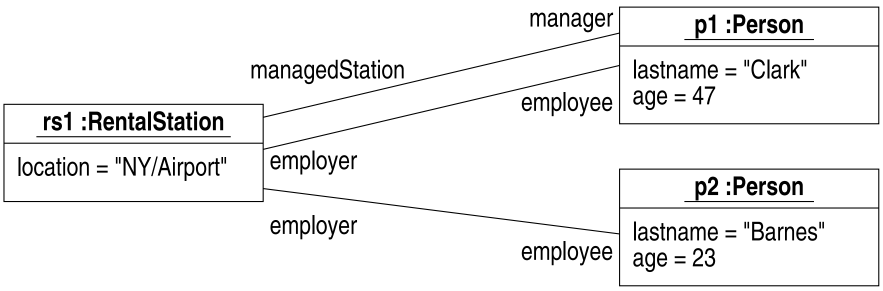

Fig. 3. Object Diagram representing a System State for $\mathcal{M}_{\mathrm{Company}}$

The definition of system state is a prerequisite for fixing the meaning of expressions referring to attributes and associations. These features are operations defined as part of a model $\mathcal{M}$. Attribute operations can be interpreted as follows: An operation $a_c : c \to s$ in $\textsc{Att}_c$ with $a_c \in \mathcal{A}_{\text{attr}}, c \in C, s \in S$ is a function mapping objects to attribute values. For a system state with graph G the semantics of a_c is given as

$$I(a_c)(\underline{c}) = \begin{cases} ostate(\underline{c})(a_c), & \text{if } \underline{c} \in V \text{ and } a_c \in \text{dom}(ostate(\underline{c})) \\ \bot, & \text{otherwise.} \end{cases}$$

The attribute *age* of a person object, for example, is given by $I(\text{age}_{Person})(p_1) = ostate(p_1)(\text{age}_{Person}) = 47$.

Finally, association operations allow the construction of navigation expressions. An n-ary association induces $n \cdot (n-1)$ operations allowing navigation among participating classes in each direction. Association ends have to be labeled by role names for this purpose. An operation $r_{<c,c'>} : c \to s$ in $\textsc{Assoc}_c$ with $r \in \mathcal{A}_{\text{role}}, c, c' \in C, s \in \{c', \mathit{Set(c')}, \mathit{Sequence(c')}\}$ maps an object of sort c either to a single object, or to a set or sequence of associated objects of sort c'.

$$I(r_{<c,c'>}) : \begin{cases} I(c) \to \mathcal{F}(I(c')) \\ \underline{c} \mapsto \{\underline{c}' \mid e \in E \land \exists i,j : \pi_i(nodes(e)) = \underline{c} \land \pi_j(nodes(e)) = \underline{c}' \\ \qquad\qquad \land \pi_j(roles(e)) = r\} \end{cases}$$

For the result, all edges being incident on the source object and target objects are considered (π_i is a function projecting a tuple to its ith element). Since there may be multiple edges (representing different associations) between the same set of objects, the edge leading to a target object must be labeled with the role name given as operation symbol. Thus for the example in Fig. 3, the interpretation of association operations yields the following results:

$I(\text{employee}_{<RentalStation,Person>})(rs_1) = \{p_1, p_2\}$
$I(\text{manager}_{<RentalStation,Person>})(rs_1) = \{p_1\}$
$I(\text{employer}_{<Person,RentalStation>})(p_1) = \{rs_1\}$
$I(\text{employer}_{<Person,RentalStation>})(p_2) = \{rs_1\}$
$I(\text{managedStation}_{<Person,RentalStation>})(p_1) = \{rs_1\}$

5 Expressions

We now define the expression syntax and its semantics for constraints and queries. First we give a complete syntax of expressions before we explain semantics and application to OCL in detail.

The definition of expressions is based upon the signature we developed in the previous sections. A signature $\Sigma_{\mathcal{M}} = (S_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$ provides a set of sorts $S_{\mathcal{M}} = S_{\text{Expr}}(S_D \cup C)$ containing basic sorts, class sorts, and complex sorts constructed by sort expressions. A partial order $\leq$ on sorts reflects subtyping. On these sorts, a set of operations $\Omega_{\mathcal{M}} = \Omega_D \cup \textsc{Att}_c \cup \textsc{Op}_c \cup \textsc{Assoc}_c$ is defined. The signature contains the initial set of syntactic elements and is basically the same as used in order-sorted algebraic specification. Based upon the signature, we define expressions.

An OCL expression is always evaluated in context of a classifier instance. Note that we distinguish between basic sorts and classifier sorts. Hence, in our framework expressions cannot be built in context of simple data values like integers.

Definition 5 (Syntax of Expressions) Let $\mathrm{Var} = \{\mathrm{Var}_s\}_{s \in S}$ be a set of variables indexed by sort s. The syntax of expressions over a classifier $c \in C$ is a set $\mathrm{Expr}(c) = \{\mathrm{Expr}(c)_s\}_{s \in S}$ indexed by sort which is defined as follows:

i. $\mathbf{self} \in \mathrm{Expr}(c)_c$.
ii. $\boldsymbol{v} \in \mathrm{Expr}(c)_s$, if $v \in \mathrm{Var}_s$.
iii. $\boldsymbol{\omega(\tau_1, \ldots, \tau_n)} \in \mathrm{Expr}(c)_s$, if $\omega : s_1 \times \cdots \times s_n \to s \in \Omega$ and $\tau_i \in \mathrm{Expr}(c)_{s'_i}$, where $s'_i \leq s_i$ for $i = 1, \ldots, n$.
iv. $\textbf{if } \boldsymbol{\varphi} \textbf{ then } \boldsymbol{\tau_1} \textbf{ else } \boldsymbol{\tau_2} \textbf{ fi} \in \begin{cases} \mathrm{Expr}(c)_{s_1}, & \text{if } s_2 \leq s_1 \\ \mathrm{Expr}(c)_{s_2}, & \text{if } s_1 \leq s_2 \end{cases}$
if $\varphi \in \mathrm{Expr}(c)_{\mathrm{Boolean}}$ and $\tau_1 \in \mathrm{Expr}(c)_{s_1}, \tau_2 \in \mathrm{Expr}(c)_{s_2}$.
v. $\boldsymbol{(\tau} \textbf{ as } \boldsymbol{s')} \in \mathrm{Expr}(c)_{s'}$, if $\tau \in \mathrm{Expr}(c)_s$ and $s' \leq s$.
vi. $\textbf{iterate}\boldsymbol{(v_1 : \tau_1, v_2 : \tau_2, \tau_3)} \in \mathrm{Expr}(c)_{s_2}$, if $v_1 \in \mathrm{Var}_{s_1}, v_2 \in \mathrm{Var}_{s_2}$, $\tau_2, \tau_3 \in \mathrm{Expr}(c)_{s_2}$, and
$\tau_1 \in \mathrm{Expr}(c)_{Sequence(s_1)} \cup \mathrm{Expr}(c)_{Set(s_1)} \cup \mathrm{Expr}(c)_{Bag(s_1)}$.

A self expression (i) provides a reference to a classifier the instance of which giving a context for evaluation. Variable bindings (ii) are introduced in features which are based on the *iterate* abstraction (`select`, `exists`, etc.) [24]. Operation expressions (iii) include predefined operations for e.g. integer arithmetic as well as for accessing object attributes and associations. Alternatively, an operation expression may also be written in OCL path syntax as $\tau_1.\omega(\tau_2, \ldots \tau_n)$. If τ_1 denotes a collection value the dot may be replaced by an arrow symbol: $\tau_1 \to \omega(\tau_2, \ldots \tau_n)$. An *if* expression (iv) provides an alternative selection of two expressions. An *as* expression (v) can be used in cases where static type information is insufficient. It corresponds to the `oclAsType` feature in OCL. The iterate feature of OCL is captured by *iterate* expressions (vi). A large group of query constructs can be reduced to *iterate* expressions [24].

The semantics of each kind of expression is fixed in the following definition. A context for evaluation is given by a classifier instance $\underline{c} \in I(c)$ and a variable assignment $\beta : \mathrm{Var}_s \to I(s)$.

Definition 6 (Semantics of Expressions) Let the set of all variable assignments be B. The semantics of an expression $e \in \mathrm{Expr}(c)_s$ is a function $I[\![\, e \,]\!] : B \times C \to I(s)$ defined by induction on the structure of e:

i. $I[\![\, \mathrm{self} \,]\!](\beta, \underline{c}) = \underline{c}$.
ii. $I[\![\, v \,]\!](\beta, \underline{c}) = \beta(v)$.
iii. $I[\![\, \omega(\tau_1, \ldots, \tau_n) \,]\!](\beta, \underline{c}) = I(\omega)(I[\![\, \tau_1 \,]\!](\beta, \underline{c}), \ldots, I[\![\, \tau_n \,]\!](\beta, \underline{c}))$.
iv. $I[\![\, \text{if } \varphi \text{ then } \tau_1 \text{ else } \tau_2 \text{ fi} \,]\!](\beta, \underline{c}) = \begin{cases} I[\![\, \tau_1 \,]\!](\beta, \underline{c}), & \text{if } I[\![\, \varphi \,]\!](\beta, \underline{c}) = \text{true} \\ I[\![\, \tau_2 \,]\!](\beta, \underline{c}), & \text{if } I[\![\, \varphi \,]\!](\beta, \underline{c}) = \text{false} \\ \bot, & \text{otherwise.} \end{cases}$

v. $I[\![(\tau \text{ as } s')]\!](\beta, \underline{c}) = \begin{cases} I[\![\tau]\!](\beta, \underline{c}), & \text{if } I[\![\tau]\!](\beta, \underline{c}) \in I(s') \\ \bot, & \text{otherwise.} \end{cases}$

vi. $I[\![\text{iterate}(v_1 : \tau_1, v_2 : \tau_2, \tau_3)]\!](\beta, \underline{c}) = I[\![\text{iterate}'(v_1 : \tau_1, \tau_3)]\!](\beta', \underline{c})$ where β' is a modified assignment $\beta' = \beta\{v_2/I[\![\tau_2]\!](\beta, \underline{c})\}$.

(a) if $\tau_1 \in \text{Expr}(c)_{Sequence(s_1)}$ then $I[\![\text{iterate}'(v_1 : \tau_1, \tau_3)]\!](\beta, \underline{c}) =$
$$\begin{cases} I[\![v_2]\!](\beta, \underline{c}), & \text{if } I[\![\tau_1]\!](\beta, \underline{c}) = <> \\ I[\![\text{iterate}'(v_1 : \text{tail}(\tau_1), \tau_3)]\!](\beta'', \underline{c}), & \text{if } I[\![\tau_1]\!](\beta, \underline{c}) = < e_1, \dots, e_n > \end{cases}$$

(b) if $\tau_1 \in \text{Expr}(c)_{Set(s_1)} \cup \text{Expr}(c)_{Bag(s_1)}$ then $I[\![\text{iterate}'(v_1 : \tau_1, \tau_3)]\!](\beta, \underline{c}) =$
$$\begin{cases} I[\![v_2]\!](\beta, \underline{c}), & \text{if } I[\![\tau_1]\!](\beta, \underline{c}) = \emptyset \\ I[\![\text{iterate}'(v_1 : \tau_1 - e_1, \tau_3)]\!](\beta, \underline{c}), & \text{if } e_1 \in I[\![\tau_1]\!](\beta, \underline{c}) \end{cases}$$

with β'' being a modified assignment $\beta'' = \beta\{v_2/I[\![\tau_3]\!](\underline{c}, \beta\{v_1/e_1\})\}$.

In definition vi.(a) we used the notation tail(τ_1) for an operation that, when applied to a sequence, returns a sequence with its head element removed. We could also use the OCL operation τ_1`->subSequence(2,`τ_1`->size)` for this purpose. Also, in definition vi.(b) we wrote $\tau_1 - e_1$ for an operation that removes the element e_1 from a set or bag given by the expression τ_1. For sets the corresponding OCL feature is τ_1`->excluding(`e_1`)`. For bags however, we can not use the equally named OCL feature since it removes *all* occurrences of e_1 at once which is not what we want here.

An iterate term effectively receives three parameters: a collection S (set, sequence or bag, given by the term τ_1) containing the elements $e_1, \dots, e_n$, an initial value a (given by the term τ_2), and a binary operation $\oplus$ (given by τ_3). The semantics of an iterate term is then given by the expression $(((a \oplus e_1) \oplus e_2) \cdots \oplus e_{n-1}) \oplus e_n$. This kind of evaluation is commonly known in functional style programming languages as *fold* operation.

Note however, that we do not restrict application of iterate expressions to collections of type Sequence. For sets and bags we cannot assume any particular order in which elements of the collection are selected for applying $\oplus$. Therefore, we require that $\oplus$ has the property $(a \oplus x_i) \oplus x_j = (a \oplus x_j) \oplus x_i$ for any two elements $x_i, x_j \in S$. If $\oplus$ is associative and commutative then this property is fulfilled. For operations where evaluation order makes a difference (e.g., string concatenation) a non-ordered collection first has to be transformed into a sequence before iterate can be applied.

6 Summary and Conclusion

The Object Constraint Language is an important extension to the UML language. Many kinds of constraints that cannot be expressed purely in the more graphically oriented UML notation can be elegantly stated with OCL expressions. We argued that a formal language also should have a formal semantics and presented our approach to developing a precise semantics of OCL.

We identified basic syntactic structures and assigned to them a semantics based on set theory. We also tried to solve some of the problems inherent to the current OCL definition. For example, we proposed a solution to the problems involved in the implicit flattening of complex results, and we pointed out some non-obvious properties of operations which are preconditions for an unambiguous interpretation of iterate expressions. With these rules, the example given in the introduction can be classified as being illegal since the operation `concat` does not have the property $(a \texttt{ concat } x_i) \texttt{ concat } x_j = (a \texttt{ concat } x_j) \texttt{ concat } x_i$. To correct the query, the iteration target first must be explicitly transformed from a set to an ordered sequence.

In our formalization approach, we strictly distinguish between (immutable) data types and object types having mutable state. This distinction also helps in determining expressions which are safe with respect to termination of evaluation and finite results. This is an important aspect if we want to build CASE tools supporting simulation and validation of models. Most OCL expressions refer to elements in concrete UML class models. This requires a well-defined interface to class models and a precise understanding of the semantics of classes, associations, attributes, and generalization. As far as we know, a comprehensive and precise UML semantics is still not available. Therefore we had to introduce a concept for formally representing class models and their intended meaning by the notion of system state. Future work could extend this framework of formalization to other aspects of UML.

Acknowledgments

We thank the anonymous referees for their helpful comments.

References

1. M. Aksit and S. Matsuoka, editors. *ECOOP'97 — Object-Oriented Programming: 11th European Conference, Jyvaskyla, Finland, June 9–13, 1997: Proceedings*, LNCS 1241. Springer-Verlag, 1997.
2. E. Bertino, D. Castelli, and F. Vitale. A Formal Representation for State Diagrams in the OMT Methodology. In K. G. Jeffery, J. Kral, and M. Bartosek, editors, *Proc. Seminar Theory and Practice of Informatics (SOFSEM'96)*, LNCS 1175, pages 327–341. Springer-Verlag, 1996.
3. J. C. Bicarregui, K. Lano, and T. S. E. Maibaum. Objects, Associations and Subsystems: A Hierarchical Approach to Encapsulation. In Aksit and Matsuoka [1], pages 324–343.
4. G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 1994.
5. R. Bourdeau and B. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, 1995.
6. R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a Formalization of the Unified Modeling Language. In Aksit and Matsuoka [1], pages 344–366.

7. R. G. G. Cattell, editor. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, Inc., 1997.
8. P. P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Trans. on Database Systems*, 1(1):9–36, 1976.
9. J. Ebert and R. Süttenbach. Integration of Z-Based Semantics of OO-Notations. In Kilov and Rumpe [17], pages 75–81.
10. A. Evans, R. France, K. Lano, and B. Rumpe. Developing the UML as a Formal Modelling Language. In Muller and Bézivin [20], pages 297–307.
11. R. France, J. M. Bruel, M. Larrondo-Petrie, and M. Shroff. Exploring the Semantics of UML type structures with Z. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 247–260. Chapman and Hall, London, 1997.
12. R. France, A. Evans, and K. Lano. The UML as a Formal Modeling Notation. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Proc. OOPSLA'97 Workshop on Object-oriented Behavioral Semantics.* TU München, TUM-I9737, 1997.
13. M. Gogolla. *An Extended Entity-Relationship Model – Fundamentals and Pragmatics.* LNCS 767. Springer-Verlag, Berlin, 1994.
14. M. Gogolla and M. Richters. On Constraints and Queries in UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications.* Physica-Verlag, Heidelberg, 1998.
15. A. Hamie, J. Howse, S. Kent, R. Mitchell, and F. Civello. Reflections on the OCL. In Muller and Bézivin [20], pages 137–145.
16. I. Jacobsen, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.
17. H. Kilov and B. Rumpe, editors. *Proc. of the ECOOP'97 Workshop on Precise Semantics for Object–Oriented Modeling Techniques, Jyväskylä, Finland, 10 June 1997.* TU München, TUM-I9725, June 1997.
18. A. Kleppe, J. Warmer, and S. Cook. Informal formality? The Object Constraint Language and its application in the UML metamodel. In Muller and Bézivin [20], pages 127–136.
19. K. Lano. Enhancing Object-Oriented Methods with Formal Notations. *Theory and Practice of Object Systems*, 2(4):247–268, 1996.
20. P.-A. Muller and J. Bézivin, editors. *Proc. of UML'98 International Workshop, Mulhouse, France, June 3 - 4*, 1998.
21. G. Övergaard and K. Palmkvist. A Formal Approach to Use Cases and their Relationships. In Muller and Bézivin [20], pages 309–317.
22. Rational Software Corporation, Unified Modeling Language (UML) version 1.1, `http://www.rational.com`, 1997.
23. UML Semantics, 1997. (Published as part of [22]).
24. Object Constraint Language Specification, 1997. (Published as part of [22]).
25. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs (NJ), 1991.
26. M. Shro and R. B. France. Towards a Formalization of UML Class Structures in Z. In *Proc. 21st Annual Int. Computer Software and Applications Conference (COMPSAC'97)*, pages 646–651. IEEE, 1997.
27. E. Y. Wang, H. A. Richter, and B. H. C. Cheng. Formalizing and integrating the object and dynamic models within OMT. In *Proc. 19th Int. Conf. on Software Engineering (ICSE'97)*, pages 45–55. ACM Press, 1997.
28. J. Warmer, J. Hogg, S. Cook, and B. Selic. Experience with Formal Specification of CMM and UML. In Kilov and Rumpe [17], pages 167–171.