A Precise Approach to Validating UML Models and OCL Constraints

Mark Richters
Fachbereich 3
Universität Bremen

Draft version 0.215

Abstract

The Unified Modeling Language (UML) is a widely accepted standard for modeling software systems. The UML supports object-oriented approaches to software development with a rich set of modeling concepts. The graphical notation of UML includes diagrams such as use case diagrams, class diagrams, state diagrams and sequence diagrams. These are used for describing static as well as dynamic aspects of a system. An important part of UML is the Object Constraint Language (OCL) – a textual language that allows to specify additional constraints on models in a more precise and concise way than it is possible to do with diagrams only. While UML offers a rich set of concepts and diagrams, it is still an unsolved problem what the precise meaning of a model and associated constraints is. A number of problems related to under-specified constructs, ambiguities and contradictions have already been identified in the past. In our view, it is important to have a precise semantics of UML models and OCL constraints. Precise foundations are needed for analysis, validation, verification, and transformation (such as refinement and code generation) of models. They are also a prerequisite for providing tools with a well-defined and predictable behavior.

We present a precise approach that allows an analysis and validation of UML models and OCL constraints. We focus on models and constraints specified in the analysis and early design stage of a software development process. For this purpose, a suitable subset of UML corresponding to information that is usually represented in class diagrams is identified and formally defined. This basic modeling language provides a context for all OCL constraints. We define a formal syntax and semantics of OCL types, operations, expressions, invariants, and pre-/postconditions. We also give solutions for problems with the current OCL definition and discuss possible extensions. A metamodel for OCL is introduced that defines the abstract syntax of OCL expressions and the structure of types and values. The metamodel approach allows a seamless integration with the UML metamodeling architecture and makes the benefits of a precise OCL definition easier accessible. The OCL metamodel also allows to define context-sensitive conditions for well-formed OCL expressions more precisely. These conditions can now be specified with OCL whereas they previously were specified only informally. In order to demonstrate the practical applicability of our work, we have realized substantial parts of it in a tool supporting the validation of models and constraints. Design specifications can be "executed" and animated thus providing early feedback in an iterative development process. Our approach offers novel ways for checking user data against specifications, for automating test procedures, and for checking CASE tools for standards conformance. Therefore, this work contributes to the goal of improving the overall quality of software systems by combining theoretical and practical techniques.

Acknowledgements

I am grateful to my supervisor Martin Gogolla for many fruitful and helpful discussions in the stimulating atmosphere of his research group. I also thank my co-supervisor Hans-Jörg Kreowski for accepting the task of examining this thesis. Furthermore, I am grateful to my colleagues Heino Gärtner, Markus Germeier, Anne-Kathrin Huge, Ralf Kollmann, Oliver Laumann, Arne Lindow, and Oliver Radfelder. I also thank Paul Ziemann and Oliver Radfelder for reading this thesis and providing helpful suggestions. I received valuable feedback that helped to improve the USE tool described in this work from many people, in particular, Jose Alvarez, Jörn Bohling, Tim Harrison, Stuart Kent, and Arne Lindow. Thanks go also to members of the OCL workshops, especially Jos Warmer and Heinrich Hussmann, for helpful discussions on all aspects of OCL, to students of the SIGN project, my colleagues from the Bremen Institute of Safe Systems (BISS), and all the people who contributed to our online bibliography on UML. Last but most important, I thank Susanne for her patience and continuous support.

Contents

Chapter 1

Introduction

Analysis and design are important tasks in the process of constructing software systems. A number of different methods and languages support systematic approaches to these tasks. The *Unified Modeling Language* (UML) has recently gained much attention in this area [BRJ98, RJB98, FS97]. UML has been developed to integrate and unify many of its popular predecessor languages including Booch [Boo94], OMT [RBP+91], and OOSE [JCJÖ92]. The language was accepted as a standard by the *Object Management Group* (OMG) in 1997. At the time of this writing, the most recent version adopted as an official standard by the OMG is UML 1.3 [OMG99c].

The UML provides nine diagram types such as use case diagrams, class diagrams, state diagrams and sequence diagrams for modeling different aspects of a system. A further important part of UML is the *Object Constraint Language* (OCL) [OMG99b, WK98, WK99] – a textual language that allows to specify additional constraints on models in a declarative way similar to predicate logic. While the UML offers an appealing and expressive notation it is still an unsolved problem what the precise meaning of a model is. A number of problems related to under-specified constructs, ambiguities and contradictions have already been identified in the past (a good source for details are the proceedings of the ≪UML≫ conference series [BM99, FR99, EKS00] and related workshops, for example, [SK98, AMDK98, FRHS+99, CKW00a, CW00]).

Although the UML definition is much more rigorous than most of its predecessors like OMT, the syntax and semantics of UML currently do not have a formal foundation. In our view, it is important to have a precise semantics of UML models and OCL constraints [RG99b]. Precise foundations are needed for analysis, validation, verification, and transformation (such as refinement and code generation) of models. They are also a prerequisite for providing tools with a well-defined and predictable behavior.

In the area of database modeling the UML has to compete with (Extended) Entity-Relationship modeling approaches which often do have a solid formal foundation [Gog94].

The goal of this thesis is to define a precise framework that allows an analysis and validation of OCL constraints in UML models. The framework is based on a core language of basic UML modeling concepts and the Object Constraint Language for the specification of constraints on models.

Formalizing the UML is a huge and difficult task. In our work we focus on those features of UML which are important for the analysis and early design phase in the software development process. These features are: (1) UML language constructs for describing structural and dynamic properties of objects, and (2) OCL for specifying integrity constraints and queries. We will not consider language constructs for requirements analysis, e.g. use cases, and for implementation aspects such as component and deployment diagrams.

Related Work

There are a number of groups working on a formalization of UML. Results have been presented on several workshops and conferences (ECOOP, OOPSLA, ≪UML≫'98-2001, among others).[1] A larger coordinated effort towards a formalization of UML is organized by the *precise UML Group* (pUML). A summary of their work can be found in [Eva98]. The compilation in [AEF+99] tries to answer questions related to a UML semantics in the style of an FAQ (Frequently Asked Questions). Specific approaches are, for example, addressing the semantics of dynamic modeling notations in UML, such as interactions and sequence diagrams, using Real-time Action Logic [LB98]. An integrated view based on a mathematical system model is proposed in [BGH+97]. Another approach focuses on the integration of UML with mature formal specification notations like Z and Object-Z [FBLPS97]. There are a number of different approaches, but yet none of them has led to a complete formal definition of UML. Therefore, some authors focus on manageable parts of UML, for example, sequence diagrams [Ara98], state diagrams [GPP98], use cases and their relationships [ÖP99], or the package concept [SW98].

There are also various different approaches in related work addressing formal aspects of OCL. A graph-based semantics for OCL was developed by translating OCL constraints into expressions over graph rules [BKPPT00]. A formal semantics was also provided by a mapping from (a subset

[1] An online bibliography with references to publications relevant for research on UML is available at [Ric00].

of) OCL to a temporal logic [DKR00], to the Larch specification language [HCH+99, HHK98a], and to algebraic specifications [BHTW99]. The expressive power of OCL in terms of navigability and computability is discussed in [MC99]. Metamodels for OCL [RG99a, BH00] follow the metamodeling approach used in the UML standard. Type checking OCL expressions is investigated in [Cla99]. The important role of OCL is also acknowledged in proposals for future UML versions. For example, in [CEK01] a Metamodelling Language including OCL is proposed as a core language for UML 2.0. Several extensions to OCL have been proposed. These include temporal operators [RM99, CT01], redundant invariants and time-based constraints [Ham99], and behavioral constraints on occurrences of events, signals, and operation calls [KW00, SS00].

We conclude that work on formalizing UML and OCL is still in its early phase. Some promising work has been done on various aspects of UML. Often, different application domains impose further special requirements. In particular, for information systems modeling the role of OCL as a constraint and query language is much more important than for modeling (for example) real-time systems. In fact, the application of OCL as a *query* language was not considered in the original language definition but has proven to be very useful for data-intensive applications.

Research Goals and Contributions

The goal of this work is to define a precise syntax and semantics for the Object Constraint Language and essential parts of the UML core that provide the context for OCL constraints. The practical benefits of this formalization are demonstrated by a tool-based approach for validating OCL constraints in UML models. We claim that a formalization of OCL and a well-defined subset of UML provides a proper foundation for precise modeling of software systems. Our work makes the following significant contributions.

- Parts of the UML core are formally defined allowing an unambiguous interpretation of essential concepts in software modeling.
- For the first time a precise definition of OCL avoiding ambiguities, under-specifications, and contradictions is given. Several lightweight extensions are proposed to improve the orthogonality of the language.
- The integration of UML and OCL is improved by making the relationships and dependencies between both languages explicit.
- A solid foundation for tools supporting analysis, simulation, transformation and validation of UML models with OCL constraints is developed. Results of this work are implemented in a tool for validating OCL constraints.

- Our approach has been used to validate the well-formedness rules in the UML standard. The results provide input for improving future UML versions.

In combination, these aspects contribute to the goal of improving the overall quality of software systems based on the availability of well-defined modeling languages. This thesis integrates and extends results that have in part been presented in [RG00c, RG00b, GRKR00, GR00, RG00d, RG99b, RG99a, GRR99a, GRR99b, AEF+99, RG98, GR98b, GR98a, GR98c].

Thesis Structure

The thesis is organized as follows. Chapter 2 gives background information on UML and OCL. It covers aspects mainly related to language definition. The need for a precise constraint language is further motivated. In Chapter 3, a basic modeling language is described that suffices for describing models with class diagrams focusing on static and structural aspects during the analysis and early design phase. This subset of UML provides the context that is necessary for applying OCL constraints and queries. Chapter 4 gives a formalization of types and operations in OCL. Syntax and semantics of expressions and constraints such as invariants and pre- and postconditions are defined in Chapter 5. The fundamental concepts presented thus far are used in Chapter 6 to define a metamodel for OCL in the same style as most of UML is defined. Chapter 7 reports on a tool-based approach to validating UML models and OCL constraints. The tool implements and therefore shows the feasibility of many of the ideas and results presented in this work. Chapter 8 concludes the thesis with a short summary and an outline of the contributions and future work.

Chapter 2

Background

This chapter presents background information on UML and OCL. We concentrate on aspects concerning the language definitions. The application of UML to building models of software systems is not discussed. For this and more detailed information, there are many text books which can be consulted, for example, [BRJ98, RJB98, JBR99, HK99, FS97].

Section 2.1 discusses the UML with respect to its language definition and its notation. An example model is introduced showing the main concepts of class diagrams. UML class diagrams provide many important features for modeling the static structure of a system. The example model is frequently used throughout the remaining text for discussing various concepts and their formalization. Section 2.2 gives an overview of OCL. The current state of OCL is discussed, and the need for a precise definition is motivated. The chapter closes with a summary.

2.1 Unified Modeling Language

Object-oriented modeling languages like OMT [RBP+91] or Booch [Boo94] are often defined only by informal explanations of their concepts and notation. This makes it difficult to get a precise and unambiguous understanding of models. At first glance, it might seem sufficient for a modeling language to enable developers to communicate ideas and decisions made during the design of a software system. However, this mainly touches the pragmatics of a language. For tasks like the refinement of a design into an implementation, and (preferably tool-based) support for analysis and transformation of models, a precise definition of the syntax and semantics of a modeling language is necessary.

Comparisons between informal languages are difficult because sometimes a different notation is used for the same concept, and sometimes a single

notation may have different interpretations in different languages. Criteria for classifying and comparing modeling languages are developed and applied to more than 50 modeling languages in [Ste97].

UML is a modeling language that differs from many of its predecessors in that a structured and formal approach has been applied for its definition. In the following, we consider the approach for defining UML in more detail.

2.1.1 Language Definition

The Unified Modeling Language is defined in a standards document published by the Object Management Group (OMG) [OMG99c]. The parts of this document that are relevant for our discussion are the chapters on UML Semantics [OMG99e], the UML Notation Guide [OMG99d], and the Object Constraint Language Specification [OMG99b]. The abstract syntax and intended semantics of UML language constructs are defined in the UML Semantics part. A concrete syntax in form of a graphical notation for diagrams is introduced in the UML Notation Guide. The concrete syntax and pragmatics of OCL are defined in the Object Constraint Language Specification.

The definition of the abstract syntax of UML follows a metamodeling approach, that is, the same technique used to model application domains is used to model the UML itself. Table 2.1 gives an overview of the general four layer metamodeling architecture of UML according to [OMG99e].

Level	Layer	Description
M3	meta-metamodel	Language for metamodels
M2	metamodel	Language for models
M1	model	Language for information domains
M0	user objects (user data)	Specific information domain

Table 2.1: UML metamodeling architecture

The M0 level represents user objects of a specific information domain. These user objects and data are described by a UML model on the M1 level. The main language constructs on the M2 metamodel layer are classes and associations describing model elements on the M1 level. The M2 level is given by the UML metamodel in [OMG99e]. The M3 level is a subset of UML and provides the language for the M2 layer. Class diagrams in the metamodel are used to define properties and relationships between model elements. The semantics of model elements is given informally as English text. In most cases, additional constraints are required as context conditions

to further restrict the set of legal UML models. These constraints are defined by well-formedness rules expressed as OCL invariants on model elements.

A UML model conforms to the OMG standard if it respects all requirements specified in the standard document. We can also define standard conformance more technically in terms of the metamodel architecture: A UML model conforms to the standard if it can be fully represented as an instance of the UML metamodel. In Chapter 7, we will see how this definition can be utilized for automatic and tool-supported validation of standard conformance.

The current approach to defining syntax and semantics of UML leads to the following problems:

- The definition of UML with UML introduces a circularity. Strictly speaking, for understanding the definition of UML, the reader already has to know the meaning of those UML constructs that are used as a definition language. In general, it is preferable to use a language with an already known semantics to define another language [Rum98, HR00]. Proposals for a formal object-oriented metamodeling approach have been made in [CEK01, Öve00].

- An informal semantics definition with English text may introduce inconsistencies and ambiguities. This has been pointed out by various authors, for example, [BHH+97, SW98].

- Model elements describing entities of different conceptual levels are defined within a single level given by the metamodel layer. For example, a class usually contains attributes. This is a structural relationship that can be adequately expressed by an association in the metamodel. On the other hand, classes specify the set of possible objects, and attributes specify the set of possible attribute values. This different kind of relationship expressing the semantics of classes and attributes is denotated in the metamodel by the same mechanism – a plain association. The metamodel provides a uniform framework for modeling all kinds of concepts and relationships. However, it draws no clear boundary between elements of different conceptual levels.

- Using OCL for specifying well-formedness rules adds a certain degree of preciseness. Warmer et al. report on significant improvements resulting from the use of OCL instead of plain text [WHCS97]: A lot of ambiguities and contradictions in an earlier version of UML could be resolved by reconsidering and reformulating the well-formedness rules with OCL. However, since OCL currently also lacks a precise semantics, the well-formedness rules may still be open to different interpretations.

Chapter 3 presents an alternative definition of UML core concepts that avoids the first three problems mentioned above. Chapters 4 and 5 provide a precise foundation for OCL, therefore tackling the last issue.

2.1.2 Notation

UML is a language with a graphical notation. Models are represented as a set of diagrams each focusing on different aspects of a design. The notation of diagram elements is defined in the UML Notation Guide [OMG99d]. The following gives a brief overview of the nine different diagram types available in UML. A more detailed classification based on views supported by each of the diagram types can be found in the Unified Modeling Language Reference Manual [RJB98].

Use Case diagrams. The functionality of a system is modeled as a set of use cases each showing a possible interaction between a user, also called an actor, and the system. The focus lies on listing actors and the use cases in which they participate. Separate text is frequently used to describe the nature of a use case in detail.

Class diagrams. The central concepts for modeling structural properties of a system are classes and their relationships like aggregation and generalization. Class diagrams focus on time-independent aspects of classes and relationships.

Object diagrams. Objects are instances of classes. An object diagram provides a snapshot of a system at a particular point in time showing objects, their attribute values, and links connecting the objects. Object diagrams are considered a subset of class diagrams.

Sequence diagrams. The dynamic behavior of a system can be explained as interactions where objects exchange messages. A sequence diagram describes a sequence of message exchanges showing the objects participating in the interaction and the order in time in which messages are exchanged.

Collaboration diagrams. Alternative representations of interactions can be given by collaboration diagrams. While sequence diagrams emphasize time aspects, collaboration diagrams are suitable for highlighting structural aspects. Links between objects indicate channels for message passing.

Statechart diagrams. The life cycle of an object can be considered a sequence of different states of a state machine associated with the object.

Statechart diagrams are an extension of the classical concept of state machines [Har87].

Activity diagrams. By interpreting the states (or nodes) of a state machine as activities, each node can be used to describe a step in a computation. Activity diagrams are a variant of statechart diagrams in which an outgoing transition of a node implicitly fires when the source node completes its activity.

Component diagrams. A system implementation consists of a number of software units. These components together with their dependencies are shown in component diagrams.

Deployment diagrams. Component instances are processed at run-time on processing nodes. A deployment diagram shows the configuration of processing nodes and component instances.

Diagram types can be arranged into groups each emphasizing a common aspect. This is visualized as a UML class diagram in Figure 2.1. Diagram types are classified either as use case, static structure, behavioral, or implementation diagram. Classes with names set in italic denote abstract concepts being a generalization of concrete diagram types. For example, class diagrams and object diagrams are commonly referred to as static structure diagrams.

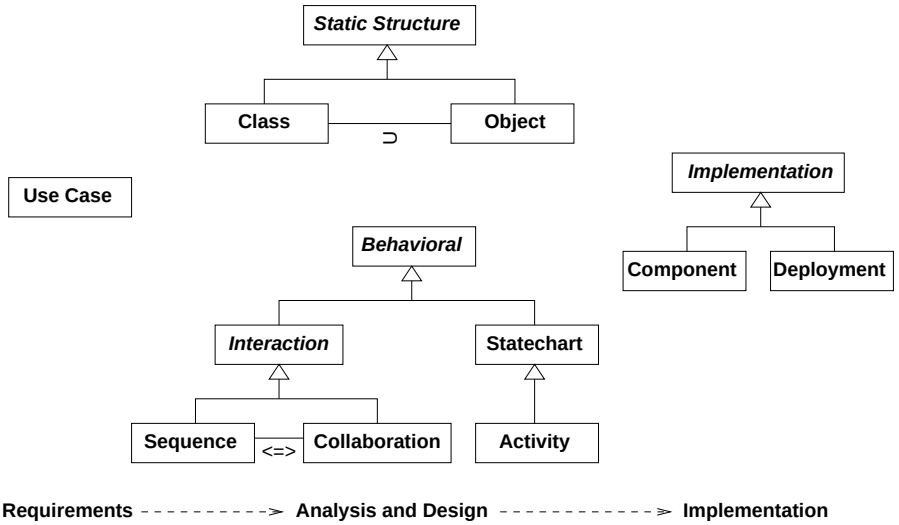


Figure 2.1: Classification of UML diagram types (based on [Gog98]). Diagram types are placed horizontally indicating their relevance to different phases of the software development process.

Sequence diagrams are equivalent to collaboration diagrams with respect to their information content; they differ only in terms of representation focusing on time or structural aspects, respectively. Both diagram types are also known as interaction diagrams. Activity diagrams are a special kind of statechart diagrams. The figure also gives a rough indication in which phases of the software development process the various diagram types are usually applied. Although a precise assignment highly depends on a concrete process, we can observe that most diagram types of UML are concerned with analysis and design. Note that many diagrams can be used to describe models at various stages of development. For example, a class diagram can present a model that is derived from an early analysis of a problem. Another class diagram may show a refined model at a later stage including information about the proposed solution to the problem as well as implementation details. A set of criteria helping to understand the differences as well as common aspects of UML diagram types can be found in [Gog98].

The advantages of a graphical modeling notation are obvious. Diagrams are easy to understand even by non-experts, and they help to structure different aspects by a visual dimension that is not available in purely textual notations.[1] Nevertheless, there are several problems related to graphical languages. One of these problems is that it still seems difficult to give a precise specification of the syntax and semantics of two-dimensional languages. The graph grammar approach may be helpful for this [Roz97]. Indeed, the application of graph transformation techniques has been suggested for giving precise semantics to UML state diagrams [GPP98] and collaboration diagrams [EHHS00], for describing a mapping from more complex UML language features to a UML core language [Gog00], for checking the consistency between UML class and sequence diagrams [TE00], and for giving a graph-based semantics to OCL [BKPPT00].

Another problematic issue with respect to the UML definition is the mapping between abstract syntax (the metamodel) and concrete syntax (the graphical notation). This mapping is defined as part of the UML Notation Guide [OMG99d]. The Notation Guide provides an informal description of how diagram elements are related to model elements. Besides the previously mentioned issues such as ambiguities and inconsistencies resulting from the informal style of this description, it is also difficult to judge whether the mapping is sound (each notational element can be mapped to a concept of the abstract syntax) and complete (each concept of the abstract syntax can be somehow represented in the notation). While there may be many notational representations of a single concept, there should, of course, exist only one well-defined mapping of a notational element to an abstract syntax element (see also [MS01] for an analysis of the visual syntax of UML).

[1] There has also been work on adding another dimension to UML diagrams resulting in 3D visualizations [GK98, GRR99a, RG00a].

2.1.3 Example Model

Throughout this text, we use an example of a case study for illustrating various concepts. The example specifies a simplified model of a car rental company. Figure 2.2 shows a UML class diagram of this model. For reasons of readability, we do not show the full signature of operations in the diagram. These can be found as part of the complete model specification in Appendix B.

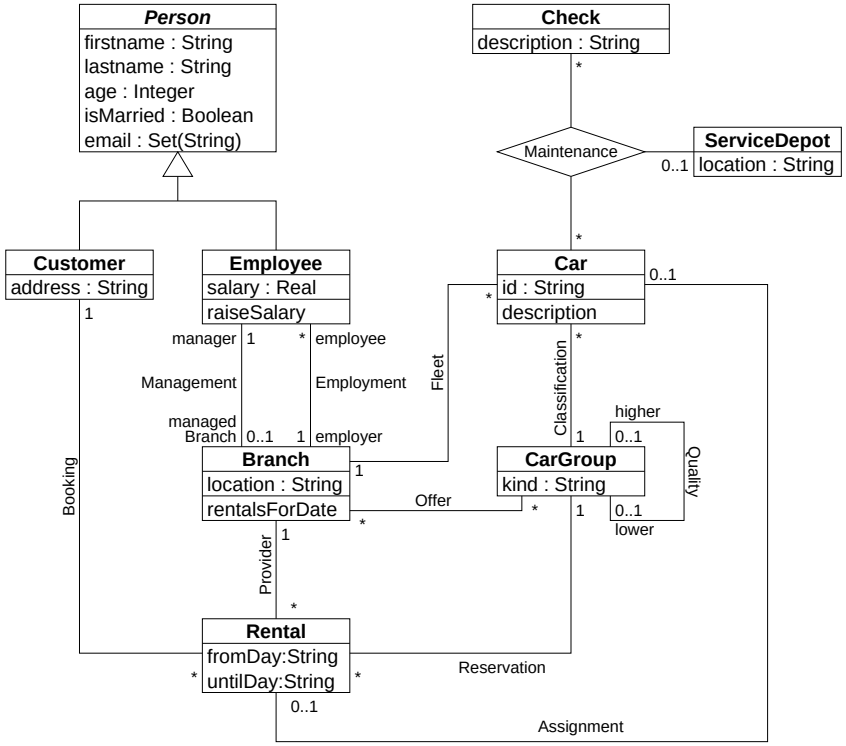


Figure 2.2: Class diagram for the car rental example

We briefly summarize the key aspects of the example. A car rental company has several branches serving customers by making reservations for cars and handling the delivery and return of cars. Each branch has a number of employees and is led by a manager. A branch owns a fleet of cars where each car belongs to a certain car group. The car group provides a classification of features available for a car. The classification scheme ranges from "economic", "compact", "intermediate" to "luxury". All cars of the same group have the same rate, but different groups usually have different rates. A branch offers car groups to customers and selects a specific car from this group only at delivery time. In a rental contract, a branch reserves a car of the desired group for a customer. When the customer appears to pick up

the car, an appropriate car is selected and assigned to the rental contract. Rental cars are periodically maintained in a service depot where several checks are performed. A car is not available for rental during maintenance.

The class diagram uses the following UML concepts: Classes (*Person*, *Branch*, *Car*, etc.) with attributes and operations, associations (Booking, Fleet, Employment, etc.), generalization (*Person* is a generalization of *Customer* and *Employee*), a ternary association (Maintenance), a self (or reflexive) association (Quality), and multiple associations between classes (Employment and Management). Association ends are adorned with role names and multiplicity specifications. A multiplicity such as `0..1` restricts the number of objects that can be linked together.

We use OCL to add further constraints to the model. Constraints are given in a textual language with references to elements of the class diagram. For example, the class diagram specifies an attribute age in class *Person* that is of type *Integer*. Without further constraints, a person object may have negative values for the age attribute. The following constraint refers to class *Person* and restricts the allowed range of age values to positive integers.

```
-- The age attribute of persons is greater than zero.
context Person inv Person1:
  self.age > 0
```

A class may have a number of constraints all of which must be satisfied for all instances of the class. We show two further examples with more complex OCL expressions to give a first impression of how constraints may look like. The details of OCL are discussed in the next section.

```
context Branch
  -- Each manager is also an employee of the branch.
  inv Branch1:
    self.employee->includes(self.manager)

  -- Managers get a higher salary than employees.
  inv Branch2:
    self.employee->forAll(e | e <> self.manager
      implies self.manager.salary > e.salary)
```

2.2 Object Constraint Language

The *Object Constraint Language* (OCL) is part of the UML standard [OMG99b, WK98]. It is a language allowing the specification of formal constraints in context of a UML model. Constraints are conditions on all states

and transitions between states of a system implementing a given model. A set of constraints therefore restricts the set of possible system states. Constraints are primarily used to express invariants of classes and pre- and postconditions of operations. An invariant is a statement about all existing objects of a class. Only single system states need to be considered for determining whether an invariant is fulfilled or not. Additionally, pre- and postconditions enable behavioral specifications of operations in terms of conditions on a previous state and a post-state after executing the operation.

The syntax style of OCL is similar to object-oriented programming languages. Most expressions can be read left-to-right where the left part usually represents – in object-oriented terminology – the receiver of a message. The language provides variables and operations which can be combined in various ways to build expressions. Frequently used language features are attribute access of objects, navigation to objects that are connected via association links, and operation calls. OCL defines a number of data types including basic types such as *Integer* and *Boolean*, as well as types for dealing with collections. The expressiveness of OCL mainly results from powerful collection operations providing set comprehension and universal and existential quantifiers. Properties of collections of objects can thus be specified in a declarative way. These language features suggest a close relationship to the first order predicate calculus.

All OCL expressions are side effect-free. Therefore, the evaluation of an expression never changes the system state. 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.

There are several places where OCL expressions can be used within a UML model. According to the metamodel described in [OMG99e, p. 2-14], 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 indirectly be related to groups of model elements. The following constraint asserts that all cars within the same car group have a unique id. Although the constraint is specified on class *CarGroup*, it indirectly restricts the possible extension of class *Car*.

```
context CarGroup inv:
  self.car->forAll(c1, c2 : Car |
    c1 <> c2 implies c1.id <> c2.id)
```

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, that is, the operation has the property "isQuery" (this property is an attribute of the UML metaclass Operation), its meaning can in principle be specified with an OCL expression. Examples of these can be found in the UML Semantics document where OCL is extensively used to define static well–formedness rules of the metamodel. Frequently needed operations are named and defined separately, thus reducing the overall length of rules. For operations with side effects, only pre- and postconditions can be specified with OCL. The generation of side effects is outside the scope of OCL.

The application of OCL is not tied to specific domains. In general, any UML design can benefit from precise constraints. Because of the descriptive nature of 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. Query support can be a very useful feature in tools allowing the user to navigate and inspect objects and data interactively. This is especially useful when dealing with large sets of objects as it is the case in database applications. Object browser with querying capabilities provide a powerful way for exploring large systems. For example, the PESTO tool is a user interface that supports browsing and querying of object databases [CHMW96]. We presented a similar approach based on hypertext documents in a web-based animator for the object specification language TROLL *light* [GR96, RG97a, RG97b].

2.2.1 Concepts

This section gives a brief overview of OCL language concepts. The primary references are the OMG standard [OMG99b] and the book by Warmer and Kleppe [WK98].

Lexical structure

The lexical structure of OCL is not explicitly defined, but it can partly be derived from the description and the grammar presented in [OMG99b]. An OCL constraint generally is a sequence of identifiers, literals, whitespace, special symbols, or comments. Identifiers are case sensitive. Some identifiers are keywords, e.g., `context`, `inv`, `post`. We skip the details for building identifiers, literals, and whitespace here. Special symbols such as "`.`", "`,`", "`(`", "`)`", "`->`" are used as punctuation symbols for composing larger syntactical structures, and for operators such as "`+`", "`-`", "`*`". Comments start with two dashes "`--`" and continue to the end of a line.

```
-- a comment
context, inv, post                    -- keywords
p,a1,Integer                          -- identifiers
3, 4.5, 'aString', true, Set{1,2} -- literals
```

Types

OCL is a typed language. Every expression has a type describing the domain of the result value and a set of applicable operations. Predefined basic types are *Boolean*, *Integer*, *Real*, and *String*. Examples for predefined operations on these types are logical operations like `and`, `or`, `not`, arithmetic operations such as `+`, `-`, `*`, and operations for string manipulation such as `concat`, and `substring`.

Enumeration types define sets of literals. Types defined in a UML model are also available in OCL. The corresponding element in the metamodel is Classifier. The most important type introduced by Classifiers are classes, data types, and interfaces. Operations on classifier types are defined as part of the UML model.

Collections of values can be described in OCL by the collection types $Set(T)$, $Bag(T)$, and $Sequence(T)$ representing mathematical sets, multisets, and lists, respectively. Collection types are parameterized by a type parameter T. The parameter T denotes the type of collection elements. For a concrete collection type, the type parameter must be instantiated, e.g., $Set(Integer)$.

Types are organized in a type hierarchy defining subtype and supertype relationships. For the basic types, there is only one subtype relationship: *Integer* is a subtype of *Real*. The collection types $Set(T)$, $Bag(T)$, and $Sequence(T)$ have a common supertype $Collection(T)$. The subtype relation is transitive, reflexive, and anti-symmetric. The rules also extend to the type parameter of collection types. If *T1* is a subtype of *T2*, then $Collection(T1)$ is a subtype of $Collection(T2)$. The type concept is discussed in more detail in Section 2.2.4.

Expressions

Literals and variables can be used as simple expressions. Expressions such as invariants written in context of a classifier may refer to an instance of the classifier by the reserved word `self`. More complex expressions can be built by means of operation calls and an if-then-else construct. The first argument (or target expression) of an operation call has a special role. It denotes the value or object the operation is applied to. This entity is usually made explicit by a notation where the target expression is followed by

a dot, the name of the operation, and possibly more arguments in parentheses. Nesting of operation calls therefore manifests in OCL syntax as a flat sequence separated by dots. Attributes of classes can be accessed in a similar way. If the target of an operation is a collection value, an arrow is used instead of a dot. Alternatively, the usual infix notation can be used for arithmetic operations.

```
23                    -- literal expression
v                     -- variable expression
self                  -- self expression
p.name()              -- operation call
p.name().substring(2) -- nested operation call
Set{1,2}->size        -- operation call on collection
3 + 2                 -- infix notation for operators
self.age              -- attribute access
if x.mod(2) = 0 then  -- if-expression
  'even'
else
  'odd'
endif
```

Context

A context declaration specifies the model element to which a constraint is attached. For invariants, the context declaration provides a classifier. For pre- and postconditions, a classifier and an operation signature is given. The context declaration may also contain variables that may appear free in the constraint expression. Examples are given below.

Invariants

Invariants are conditions that must be true during the lifetime of a system for all instances of a given classifier. The condition is specified as an expression with boolean result type.

```
-- The age attribute of persons
-- must be greater than zero
context Person inv:
  self.age > 0
```

An invariant always requires a context specifying the classifier to which the invariant is applied. In the example, the expression `self` refers to an object of class *Person*. Properties such as the `age` attribute may be referenced by using a dot notation. The semantics of an invariant requires that the

expression is true for all objects of the class given as context. Thus, the invariant can be evaluated by successively binding each object of a specified class to the `self` variable and evaluating the expression for each binding. The invariant holds if the expression is true for each binding of `self` to an existing object.

Pre- and postconditions

OCL expressions can be used to specify pre- and postconditions on operations. A precondition is a condition that has to be true before an operation is called, a postcondition has to be true after the operation has been called. The following example specifies both a pre- and a postcondition on the operation `raiseSalary` of class *Employee*.

```
-- If the amount is positive, raise the salary by
-- the given amount and return the new salary
context Employee::raiseSalary(amount : Real) : Real
  pre:  amount > 0
  post: self.salary = self.salary@pre + amount
        and result = self.salary
```

A `self` expression refers to the object to which the operation is applied. The notation `@pre` is allowed only in postconditions where it changes the evaluation context of an expression to the previous state. The expression `self.salary@pre` thus evaluates to an employee's salary before the operation `raiseSalary` has been executed. Note that a postcondition is the only place in OCL where more than a single system state can be referenced. Pre- and postconditions are discussed in detail in Chapter 5.

Queries

The literature on OCL mainly describes expressions for use within constraints. However, OCL can also be applied as a query language. A query is an expression with an arbitrary result type and no free variables. In this sense, the set of constraints is just a subset of all possible queries in OCL, namely the set of all expressions with boolean result type. Queries allow the retrieval of (possibly complex) information from a system in a declarative manner. In this way, OCL has some relationship to object–oriented query languages like OQL [CB00], or the latest SQL standard SQL:1999 [Int99], and to (E)ER models such as the original ER model [Che76], the data type based approach [dSNF79], INCOD [ABLV81], ECR [EWH85], BIER [EKTW86], ERC [PS89], or ORAC [WT91]. For example, the relationship to an EER calculus [Gog94] has been investigated more closely in [GR98c].

Special concepts

We briefly summarize other important OCL concepts.

- Navigation: An association between two classes provides a path that can be used for navigation. A navigation expression may start with an object and then navigate to a connected object by referencing the latter by the role name attached to its association end. The result is a single object or a collection of objects depending on the specified multiplicity for the association end. Navigation is not limited to a single association. Expressions can be chained to navigate along any number of associations.

- Iterate: OCL provides a general iteration construct by means of an `iterate` expression. The source of an iterate expression always is a collection value. For each element of the collection, an expression is evaluated whose result may be accumulated in a result variable. Many other collection operations can be defined in terms of an iterate expression.

- Undefined values: An expression may result in an undefined value. In general, an expression evaluates to an undefined value if one of its subexpressions is undefined. Exceptions to this rule are given for logical operations.

- Flattening: OCL tries to avoid complex collections by an "automatic flattening" mechanism. Whenever a nested collection occurs, it is reduced to a simple collection. Nested collections often result from navigation.

- Special types: The types *OclType*, *OclAny*, *OclExpression*, and *OclState* are special in OCL. *OclType* introduces a meta-level by describing as instances the set of all types. *OclAny* is the top of the type hierarchy for all types except for the collection types. *OclExpression* seems to be required only for defining expressions which combine other expressions. *OclState* is used for referring to state names in a state machine.

- Shorthand notations: There are several features on the syntax level for writing constraints in a more convenient and abbreviated way. A `let`-construct can be used to bind a complex expression to a variable which may then repeatedly occur in a constraint. Further shorthand notations exist for `collect` expressions and certain kinds of navigation expressions.

2.2.2 Applications

Applications of OCL can be found in several standards published by the OMG. The following list gives some examples showing applications of OCL within metamodeling frameworks for defining languages.

- The probably largest published application of OCL is within the UML standard itself. OCL is used to specify well-formedness rules for the UML abstract syntax. There are more than 150 invariants defined on the metamodel. Chapter 7 shows how these constraints can be automatically validated by a tool developed as part of this work to check UML models for conformance with the OMG standard.

- The *Meta Object Facility* (MOF) takes the metamodeling approach one step further and provides a meta-metamodel for describing metamodels in various domains [OMG99a]. For example, the UML can be considered an instance of the MOF. Because defining a metametamodel is conceptually similar to defining a metamodel, OCL is applied in a similar way within the MOF for specifying well-formedness rules.

- The *XML Metadata Interchange* (XMI) provides a mechanism for interchange of metadata between UML based modeling tools and MOF based metadata repositories in distributed heterogeneous environments [OMG99f]. OCL is used in the XMI proposal to define the XMI stream production rules. The production rules specify how a model can be transformed into an XML document conforming to the XMI proposal.

Case studies with considerable OCL portions are presented in [PHK+99, WK98]. Applications of OCL can also be found in other domains. For example, an object definition language with Quality of Service (QoS) support is presented in [Aag98]. OCL is used to specify the QoS characteristics an object provides and requires in multimedia systems. An extension of UML for modeling real-time reactive systems is given in [Mut00]. There, OCL is used to define well-formedness rules for the abstract model. These rules are then translated into the specification language of PVS (Prototype Verification System). A combination of design patterns with OCL constraint schemata is pursued in the KeY project [BHSS00, Baa00] where the commercial CASE tool TogetherJ is extended to integrate formal methods approaches with conventional UML modeling.

2.2.3 Tools

There are several tasks related to OCL for which tool support seems beneficial. For example, syntax checking of constraints helps in writing syntactically correct expressions. The next step could be an interpreter enabling the evaluation of expressions. Given a snapshot of a system, it could check the correctness of the snapshot with respect to the constraints. An alternative way for checking constraints is based on code generation. OCL expressions are transformed into statements of the implementation language. The generated code is responsible for detecting constraint violations.

A comprehensive list enumerating the most important kinds of tools supporting OCL is given in [HDF00]. The authors distinguish between tools doing (1) syntactical analysis, (2) type checking, (3) logical consistency checking, (4) dynamic invariant validation, (5) dynamic pre-/postcondition validation, (6) test automation, and (7) code verification and synthesis. The following (incomplete) list gives an overview of some OCL tools.

- Probably the first available tool for OCL was a parser developed by the OCL authors at IBM (and now maintained at Klasse Objecten). The parser is automatically generated from the grammar given in [OMG99b].
- An OCL toolset is being developed at the TU Dresden [HDF00]. Part of the toolset is an OCL compiler [Fin00] that also has been integrated with the open source CASE tool Argo/UML [R+01].
- An OCL interpreter is described in [Wit00]. It is based on our OCL metamodel described in [RG99a] and Chapter 6.
- A commercial tool named ModelRun [Bol00] provides validation of invariants against snapshots.
- The USE tool [RG00c, Ric01] allows validation of OCL constraints by checking snapshots of a system. The tool also provides support for the analysis of constraints. The ideas on which the tool is based are described in Chapter 7.

Table 2.2 compares the tools with respect to the features they support. The table only gives a rough indication about what is provided by a specific tool. However, what can clearly be seen is that logical consistency checking and code verification are features that currently none of the tools we considered here offers.

Feature	Tool				
	IBM Parser	Dresden OCL Toolkit	TU Munich	ModelRun	USE
(1) syntactical analysis	•	•	•	•	•
(2) type checking	–	•	•	•	•
(3) logical consistency checking	–	–	–	–	–
(4) dynamic invariant validation	–	–	•	•	•
(5) dynamic pre-/post-condition validation	–	–	–	–	•
(6) test automation	–	–	–	–	•
(7) code verification and synthesis	–	–	–	–	–

Table 2.2: Some OCL tools and the features they support.

2.2.4 Critical Assessment

OCL is intended to facilitate the specification of model properties in a formal yet comprehensible way [WHCS97]. 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.

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 and undefined values, and some technical issues regarding the notation and different ways of interpreting expressions. As it will turn out, there are some aspects, which are not sufficiently well-defined in the OCL specification to give a precise idea of their meaning. Some of these issues have first been identified in [GR98c]. More reports on OCL issues can be found in [CKM+99a, HCH+99, HHK98b, KWC99, Baa00]. Some of the proposals – in particular those from the "Amsterdam Manifesto on OCL" [CKM+99a] – have already been integrated in OCL 1.3. Others are scheduled for inclusion in later versions, and some need further discussion.

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*, *OclExpression*, and *OclState*.
- Additional types are introduced by model elements such as classes, interfaces, associations and enumerations being part of a UML model.

A subtype relationship induces a hierarchy on these types (see Figure 2.3). The type *OclAny* is the supertype of all other types except for the collection 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 operations for arithmetic operations (like addition and multiplication), and the type *Set* has features for common operations on sets like union, intersection, and difference.

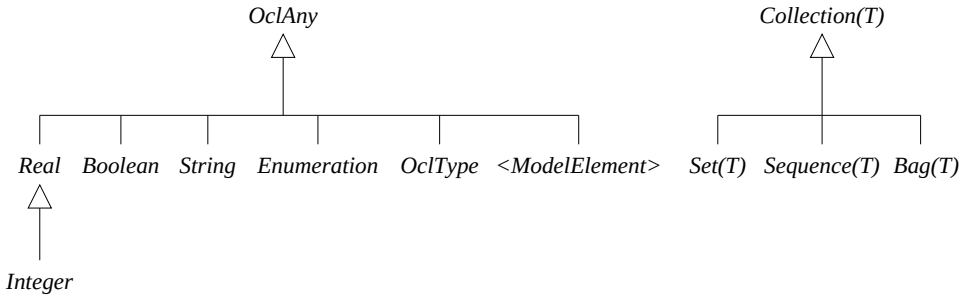


Figure 2.3: Overview of types in OCL

Operations of a specific type are also available for any of its subtypes. For example, to determine the size of a set, sequence, or bag, the operation `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 wherever 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 [Bee90]. The terms value/object and subtype/subclass are not used consistently throughout the OCL specification document. The type hierarchy as shown in Figure 2.3 has been derived from the textual description of OCL types. It is not clear whether the basic data types (*Real*, *Integer*, *Boolean*, and *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 equality operation defined for type *OclAny*.

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

Equality is an operation 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'` is type correct which probably is not the intended behavior.

The basic types also inherit some operations which refer to the state of an object (`oclInState` and `oclIsNew`). These are meaningless for value types.

The language introduces various kinds of polymorphism. We classify these according to [CW85].

- 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.

OCL defines special types *OclAny*, *OclType*, *OclExpression*, and *OclState*. The type *OclType* provides access to the meta-level of a model. This feature is rather limited since operations like `attributes` and `associationEnds` only return sets of strings. Also, it is unclear what these operations should return when they are not applied to object types but to value types.

The type *OclExpression* is a higher-order construct used for expressions that are passed as arguments to collection operations like `select`, `collect`, and `forAll`. 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?

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 parameter. The expression $Set(T)$, where T may be replaced by any OCL type, should yield a new set type with elements of type T. However, OCL does not allow the usage of collection types for T. This means that one cannot create nested collections, e.g., $Bag(Set(Integer))$ is not allowed. The problem with this restriction is that occurrences of nested values cannot be completely avoided in non–trivial models. For example, the following expression[2] determines all customers that have rented a car at a given branch (see Figure 2.2 on page 11).

```
context Branch:
  self.rental->collect(r : Rental | r.customer)
```

The result type should be $Bag(Set(Customer))$. However, according to the OCL specification the result gets "automatically" flattened into a value of type $Bag(Customer)$. How this flattening works is described only by means of the following example [OMG99b, p. 7-20]:

```
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 this collection `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, 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 with a well-defined semantics have to be defined and used.

Undefined values

Another problematic issue is the handling of undefined values. An undefined value may result from an attribute application to an object instance when

[2] We omit the keyword `inv` at the end of a context declaration if an OCL expression is not used as an invariant but as a general expression with context.

the attribute is not set or from partially defined operations 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

```
if true then
  1
else
  1 div 0
endif
```

should result in `1` whereas it is `undefined` according to the OCL rule given above. Also, it is not clear how an undefined result is handled in a context where a boolean value is expected. For example, what happens if an invariant results in undefined? Is the result then treated as `false` or as `true`? How is the concept of undefined expressions related to the universal and existential quantifier? Consider the following expressions.

```
aCollection->forAll(<expression>)
aCollection->exists(<expression>)
```

What happens in these cases if the application of `<expression>` to the elements of the collection yields undefined? According to the standard rule both expressions are undefined if `<expression>` is undefined for at least one collection element. However, if there is also an element making `<expression>` true, the exists expression should still be true as a whole.

Structured values

For the construction of new types, 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 [BOS91, Deu91, LLOW91, CB00]. This is necessary for delivering structured or aggregated query results, e.g., for retrieving a list of customer names together with the cars they have rented. Currently, this kind of query is not supported by OCL.

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 are grouped by a special argument that is – in object-oriented terms – called the "receiver" of a message. 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 in the standard document.

Resolved issues

The discussion of problematic issues with previous OCL versions has led to some improvements in the current version 1.3.

- The operation `oclType` resulting in the type of an object has been removed since an object actually can have more than one type. Information about the dynamic type of an object is frequently needed in object-oriented languages with polymorphism. In OCL – without the operation `oclType` – the dynamic type of an object can still be checked with the operations `oclIsKindOf` and `oclIsTypeOf`.

- The type system has been simplified by removing the subtype relationship between *OclAny* and the collection types. In particular, collection types with *OclAny* as element type such as $Set(OclAny)$ are not any longer subtypes of *OclAny*.

- The operation `allInstances` could be applied to any type. For classes the result is a finite set of objects existing in a given system state. However, for data types like *Integer* and *Real* the result is unclear. Consider, for example, the expression

  ```
  Integer.allInstances->size  -- Result: ???
  ```

  The use of `allInstances` on types with infinite domain is therefore discouraged in OCL 1.3.

Conclusions

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. In subsequent chapters, we propose solutions to most of the problems related to typing, flattening, undefined, and structured values.

2.2.5 Related Languages

In this section, we give a brief overview of constraint languages found in other modeling languages that are related to OCL. One of the most influencing predecessors of UML is probably OMT [RBP+91]. Constraints can be part of an OMT model. In a diagram, a condition is usually written in braces and graphically connected with the model element to be constrained. Guard expressions are used to control transitions between states in statechart diagrams. The language for constraints is not specified in OMT but can be freely chosen by the designer. Often, natural language is used in an informal way.

OMT is also the basis for a method targeting the domain of database applications as described in [BP97]. In this method, an *Object Navigation Notation* (ONN) is introduced providing a declarative way for navigating object models. The combination of ONN with pseudocode results in "enhanced pseudocode" which can be used to describe operations being part of the OMT functional model. A BNF grammar defines the syntax of ONN but leaves out the part for building boolean filter expressions. The combination with pseudocode makes it an informal language.

Syntropy is a method for object-oriented analysis and design which combines informal and formal approaches to building software [CD94]. It is based mainly on the popular graphical notations of OMT and Statecharts. For formal descriptions the basic notations of Z are used to describe sets and their properties. Syntropy makes a clear distinction between objects, values, and types. It supports various kinds of constraints, pre- and postconditions, and navigation expressions.

Catalysis has a strong emphasis on component-based development [DW98]. The notation of Catalysis is based on UML. Constraints play an important role in Catalysis. OCL is used for specifying postconditions on actions and for specifying static invariants.

Alloy is a specification language based on relations and sets. The language is used by the Alcoa tool for analyzing object models [JSS00]. The notation

of Alloy is based on Z. In [VJ99], the authors compare Alloy with OCL and give translations for a subset of the UML metamodel with well-formedness rules into Alloy.

The Extended Entity-Relationship (EER) model presented in [GH91, Gog94] provides a language and a calculus for describing database designs. 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. The EER calculus has a completely worked out formal semantics based on set theory, and equivalence rules for EER calculus expressions are known. 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 first version of OCL this was not true, for example, `Integer.allInstances` returns the infinite set of all integer values. 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 transformations. 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. A comparison of the EER approach and OCL based on an example model can be found in [GR98c].

Summary

In this chapter, an overview and a discussion of the UML and OCL language definitions were given. The metamodeling approach to defining UML provides a more precise definition than previous modeling languages could achieve with purely textual and graphical explanations. Nevertheless, it is still a compromise between formality and the goal of understandability.

OCL was introduced by explaining its basic concepts and language structure. While the UML makes a clear distinction between abstract syntax and notation, there is no such distinction in OCL. Only a concrete syntax is defined. Examples for OCL applications were given and general tool support was discussed. The introduction was followed by a critical assessment of some OCL features. A short review of similar constraint languages was given.

The next chapter defines a precise notion of an object model. The object model provides the context for most OCL expressions. Before we can give precise meaning to OCL, we therefore have to look at the object model first.

Chapter 3

Static Structure Modeling

In this chapter, the notion of an *object model* is formally defined. In our context, an object model only includes information about structural aspects of a system. It may be part of a larger model also providing behavioral specifications, but this will not be discussed here. An object model uses only those UML concepts which are essential for modeling aspects related to the structure of the problem domain during the analysis and early design phase of a development process. These concepts provide the context for OCL constraints specified on the model. Therefore, a precise understanding of object models is required before a formal definition of OCL can be given. The definition of OCL presented in Chapters 4 and 5 is based on this framework.

This chapter is structured as follows. Section 3.1 briefly describes the fundamental concepts provided by UML for modeling structural aspects. Section 3.2 discusses in more detail where these concepts are defined in the UML metamodel. Elements from the Core package of the UML metamodel are selected for a *Basic Modeling Language*. The selection process is guided by the motivation to include only those concepts that are relevant to model aspects of a given problem domain during analysis and early design phases. Thus, elements used for specifying implementations are not considered. Section 3.3 proceeds with a formal definition of the syntax of object models. The semantics of object models is defined in Section 3.4. This section also defines the notion of system states as snapshots of a running system. An alternative representation of the formal concepts is given in Section 3.5. A UML class diagram is used to visualize the main concepts. The chapter closes with a discussion that compares key aspects of our approach with a formal Extended Entity-Relationship approach.

3.1 UML Concepts for Static Structure Modeling

The fundamental concepts of UML for describing structural aspects of a system are objects and their relationships. Objects commonly represent real world entities (such as employees, cars, etc.), but they can also describe abstract concepts like purchase or reservation. Structural properties of an object are described as a set of attributes, whereas the behavior is usually encapsulated in a set of operations. The concept of a class describes a collection of objects having the same set of properties. The class is an abstraction of individual objects and specifies properties which can be observed for all objects of a given class.

Another important modeling concept in object-oriented modeling languages like UML is generalization. A generalization hierarchy organizes classes in a directed acyclic graph. Common properties of a set of classes can be placed in a generalized parent class which inherits these properties to its child classes. Generalization is a powerful concept enhancing the reusability and modularity of designs. Closely related to generalization is the notion of inheritance mainly known from object-oriented programming languages (see, e.g., [Tai96] for a comprehensive overview).

Relationships between objects are described by associations between classes. Instances of an association are called links. Classes and their relationships are graphically modeled in UML with class diagrams. Objects together with their attribute values and links between objects can be drawn as object diagrams. A class diagram describes the structural properties of all possible instances of a system. A specific instance can be described by an object diagram. Therefore, an object diagram focuses on a snapshot of an evolving system. Object diagrams are frequently used to give examples of characteristic states of a system.

The UML approach for modeling static structural aspects has its roots in the *Entity-Relationship* (ER) model [Che76] and its various extensions (e.g., the data type based approach [dSNF79], INCOD [ABLV81], ECR [EWH85], BIER [EKTW86], ERC [PS89], or ORAC [WT91]). In fact, the Object Modeling Technique (OMT) [RBP+91] as one of the most important predecessors of UML already was strongly influenced by the ER modeling approach. Concepts of the ER model which have direct counterparts in UML are, for example, entity (→ object), entity type (→ class), attribute (→ attribute), relationship (→ link), relationship type (→ association), and cardinality ratio (→ multiplicity).

3.2 A Basic Modeling Language

The Unified Modeling Language provides a number of concepts which are related to different aspects of a model. Many concepts can also be assigned to different phases of the modeling process. The following selection of basic concepts identifies a subset of UML elements which are fundamental for modeling structural aspects during the analysis and early design stage. This set of model elements is called *Basic Modeling Language* (BML) in the following. It provides the framework for defining OCL in a formal way in Chapters 4 and 5. It is not our goal to propose a new modeling language. BML is intended to be a subset of UML with a more precise syntax and semantics.

UML elements relevant for structural modeling are defined in the Core package of the UML metamodel [OMG99e, pp. 2-13]. Table 3.1 lists all elements (metaclasses) of the Core package.

For each model element, the table shows whether a model element is considered a part of our BML. The selection is based on two observations. First, elements which are defined as abstract classes in the metamodel are not included in the BML. These metaclasses never have direct instances in UML models. They merely serve as auxiliary metaclasses helping to structure the metamodel. Second, non-abstract model elements are selected for the BML by determining whether the element is essential for describing structural properties. Furthermore, the element must be predominantly used in the analysis and early design phase. Model elements which do not meet these criteria are, for example, Node and PresentationElement. The definition of Node is: "*A node is a run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well, and upon which components may be deployed.*" [OMG99e, p. 2-39]. Although a precise assignment of model elements to diagrams is missing in UML, it seems obvious from the definition that a Node is usually part of UML deployment diagrams. A Node concept should therefore not be necessary to model aspects of the problem domain.

The reason for PresentationElement not being selected for the BML becomes also obvious by looking at its definition: "*A presentation element is a textual or graphical presentation of one or more model elements.*" [OMG99e, p. 2-42]. This element refers to the notation of the modeling language but it is by itself not an essential part of a model.

For other elements such as Dependency or Component, it is more difficult to decide whether they should be part of BML. However, as mentioned earlier, our goal is to provide a context for OCL. Since the relationship between OCL and UML is not exactly defined in the standard, we focus on

Model element	Abstract element	BML element
Abstraction	•	–
Association	–	•
AssociationClass	–	•
AssociationEnd	–	•
Attribute	–	•
BehavioralFeature	•	–
Binding	–	–
Class	–	•
Classifier	•	–
Comment	–	–
Component	–	–
Constraint	–	•
DataType	–	•
Dependency	–	–
Element	•	–
ElementOwnership	–	–
Feature	•	–
Flow	–	–
GeneralizableElement	•	–
Generalization	–	•
Interface	–	–
Method	–	•
ModelElement	•	–
Namespace	•	–
Node	–	–
Operation	–	•
Parameter	–	•
Permission	–	–
PresentationElement	–	–
Relationship	•	–
StructuralFeature	•	–
TemplateParameter	–	–
Usage	–	–

Table 3.1: Model elements of the UML 1.3 Core package. For each element it is specified whether the element is abstract and whether it is considered relevant for the Basic Modeling Language.

those UML concepts that are commonly used in conjunction with OCL in practice.

Note that the introduction of BML as a subset of UML does not necessarily imply a limitation of expressiveness. Of course, all of UML can still be used for describing models. Only parts of a model that are directly or indirectly subject to OCL constraints must be expressed with BML, if preciseness of constraints is important. We are aware of the fact that not all modeling concepts are equally well amenable to a formal semantics with mathematical

rigor. Informal languages carry the risk of ambiguities and inconsistencies. However, a completely formal modeling language may become intractable with large real world applications. In the end, a mix of both styles seems more appropriate in general.

3.3 Syntax of Object Models

In this section, we formally define the syntax of object models. Such a model has the following components:

- a set of classes,
- a set of attributes for each class,
- a set of operations for each class,
- a set of associations with role names and multiplicities,
- a generalization hierarchy over classes.

Additionally, types such as *Integer*, *String*, *Set*(*Real*) are available for describing types of attributes and operation parameters. In the following, each of the model components is considered in detail. Examples are given illustrating the usage of each component. The examples frequently refer to the class diagram of our car rental case study in Figure 2.2 on page 11. The following definitions are combined in Section 3.3.7 to give a complete definition of the syntax of object models. For naming model components, we assume in this chapter an alphabet $\mathcal{A}$ and a set of finite, non-empty names $\mathcal{N} \subseteq \mathcal{A}^+$ over alphabet $\mathcal{A}$ to be given.

3.3.1 Types

A few "primitive types" including numbers and strings are predefined in UML [RJB98, p. 394]. The availability of other types like date and money is "system-dependent". In any case, the semantics of primitive types has to be defined outside UML. We need a precise definition here because types are frequently used in our object models. For example, the class diagram on page 11 uses the primitive types *Integer*, *Real*, *Boolean*, and *String* for attributes like age, salary, isMarried, and firstname, respectively.

Types are considered in depth in Chapter 4. For now, we assume that there is a signature $\Sigma = (T, \Omega)$ with T being a set of type names, and Ω being a set of operations over types in T. The set T includes the basic types

Integer, *Real*, *Boolean*, and *String*. These are the predefined basic types of OCL [OMG99b, p. 7-7]. All type domains include an undefined value that allows to operate with unknown or "null" values.

Operations in Ω include, for example, the usual arithmetic operations $+, -, *, /$, etc. for integers. Furthermore, collection types are available for describing collections of values, for example, $Set(String)$, $Bag(Integer)$, and $Sequence(Real)$. In our example model, the attribute email of class *Person* has the type $Set(String)$ to allow multiple email addresses for each person.

3.3.2 Classes

The central concept of UML for modeling entities of the problem domain is the class. A class provides a common description for a set of objects sharing the same properties.

Definition 3.1 (Classes)
The set of classes is a finite set of names $\textsc{Class} \subseteq \mathcal{N}$. □

Each class $c \in \textsc{Class}$ induces an *object type* $t_c \in T$ having the same name as the class. A value of an object type refers to an object of the corresponding class. The main difference between classes and object types is that the interpretation of the latter includes a special undefined value.

Example. The class diagram in Figure 2.2 defines the following set of classes.

$$\textsc{Class} = \{Branch, Car, CarGroup, Check, Customer, Employee, Person, Rental, ServiceDepot\}$$

□

3.3.3 Attributes

Attributes are part of a class declaration in UML. Objects are associated with attribute values describing properties of the object. An attribute has a name and a type specifying the domain of attribute values.

Definition 3.2 (Attributes)
Let $t \in T$ be a type. The attributes of a class $c \in \textsc{Class}$ are defined as a set $\textsc{Att}_c$ of signatures $a : t_c \to t$ where the attribute name a is an element of $\mathcal{N}$, and $t_c \in T$ is the type of class c. □

All attributes of a class have distinct names. In particular, an attribute name may not be used again to define another attribute with a different type.

$$\forall t, t' \in T : (a : t_c \to t \in \text{ATT}_c \text{ and } a : t_c \to t' \in \text{ATT}_c) \implies t = t'$$

Attributes with the same name may, however, appear in different classes that are not related by generalization. Details are given in Section 3.3.6 where we discuss generalization. The set of attribute names and class names need not be disjoint. We follow the UML convention of starting class names with an uppercase letter while attribute names start with a lowercase letter.

Example. Attributes of the class *Person* are defined by the following set.

$$\begin{aligned} \text{ATT}_{Person} = \{\ & \text{firstname} : \mathit{Person} \to \mathit{String}, \\ & \text{lastname} : \mathit{Person} \to \mathit{String}, \\ & \text{age} : \mathit{Person} \to \mathit{Integer}, \\ & \text{isMarried} : \mathit{Person} \to \mathit{Boolean}, \\ & \text{email} : \mathit{Person} \to \mathit{Set}(\mathit{String})\ \} \end{aligned}$$

□

Note that although attributes can be of any type, most often attributes have simple data types like *Integer*, *String*, or *Boolean*. Multi-valued attributes can be defined by using collection types. In our example, a person can have a number of different e-mail addresses. The type of the email attribute capable of storing these addresses is declared as $\mathit{Set}(\mathit{String})$.

UML also allows to specify multiplicities for attributes. For example, an optional attribute (possibly having a "null" value) is defined by specifying a multiplicity of `0..1` [OMG99d, p. 3-41]. Our definition of attributes allows "null" values because each type domain already includes the special undefined value with the same meaning. A multiplicity with an upper bound greater than one can be mapped to a collection type. Other multiplicities specify additional constraints on the range of allowed values which in general can also be expressed with OCL constraints.

Our formalism allows basic types, complex collection types and object types for attributes. Note however, that an attribute with an object type can also be modeled as an association to the corresponding class. This is a more natural approach in UML because associations emphasize the dependencies between classes. In case of a single object, the multiplicity at the target end of the association is then specified as `1` (or `0..1` if the connection to the object is optional).

3.3.4 Operations

Operations are part of a class definition. They are used to describe behavioral properties of objects. The effect of an operation may be specified in a declarative way with OCL pre- and postconditions. Chapter 5 discusses pre- and postconditions in detail. Furthermore, operations performing computations without side effects can be specified with OCL. In this case, the computation is determined by an explicit OCL expression. This is also discussed in Chapter 5. Here, we focus on the syntax of operation signatures declaring the interface of user-defined operations. In contrast, other kinds of operations which are not explicitly defined by a modeler are, for example, navigation operations derived from associations. These are discussed in the next section and in Chapter 4.

Definition 3.3 (Operations)
Let t and $t_1, \ldots, t_n$ be types in T. Operations of a class $c \in \text{CLASS}$ with type $t_c \in T$ are defined by a set OP_c of signatures $\omega : t_c \times t_1 \times \cdots \times t_n \to t$ with operation symbols ω being elements of $\mathcal{N}$. □

The name of an operation is determined by the symbol ω. The first parameter t_c denotes the type of the class instance to which the operation is applied. An operation may have any number of parameters but only a single return type. In general, UML allows multiple return values [RJB98, p. 371]. We do not include this feature here, since it seems to be rarely used, and there is no support for it in OCL.

Example. Operations of the classes *Car*, *Branch*, and *Employee* are defined by the sets given below. Note that operations are only partly shown in the class diagram on page 11. Their complete description can be found in Appendix B.

$$\begin{aligned} \text{OP}_{Car} &= \{\text{description} : Car \to String\} \\ \text{OP}_{Branch} &= \{\text{rentalsForDay} : Branch \times String \to Set(Rental)\} \\ \text{OP}_{Employee} &= \{\text{raiseSalary} : Employee \times Real \to Real\} \end{aligned}$$

□

3.3.5 Associations

Associations describe structural relationships between classes. Generally, classes may participate in any number of associations, and associations may connect two or more classes.

Definition 3.4 (Associations)
The set of associations is given by

i. a finite set of names $\textsc{Assoc} \subseteq \mathcal{N}$,

ii. a function associates : $\begin{cases} \textsc{Assoc} \to \textsc{Class}^+ \\ as \mapsto \langle c_1, \ldots, c_n \rangle \text{ with } (n \geq 2) \end{cases}$.

□

The function associates maps each association name $as \in \textsc{Assoc}$ to a finite list $\langle c_1, \ldots, c_n \rangle$ of classes participating in the association. The number n of participating classes is also called the *degree* of an association; associations with degree n are called n-ary associations. For many problems the use of binary associations is often sufficient. A *self-association* (or recursive association) sa is a binary association where both ends of the association are attached to the same class c such that associates$(sa) = \langle c, c \rangle$. The function associates does not have to be injective. Multiple associations over the same set of classes are possible. For example, the associations Management and Employment in Figure 2.2 both connect the classes *Employee* and *Branch*.

Example. The class diagram in Figure 2.2 has ten binary associations and one ternary association named Maintenance. The association Quality is a self-association on *CarGroup*.

$$\begin{aligned} \textsc{Assoc} = \{&\text{Assignment}, \text{Booking}, \text{Classification}, \text{Employment}, \\ &\text{Fleet}, \text{Maintenance}, \text{Management}, \text{Offer}, \\ &\text{Provider}, \text{Quality}, \text{Reservation}\} \end{aligned}$$

$$\begin{aligned} \text{associates(Assignment)} &= \langle \textit{Rental}, \textit{Car} \rangle \\ \text{associates(Booking)} &= \langle \textit{Rental}, \textit{Customer} \rangle \\ \text{associates(Classification)} &= \langle \textit{CarGroup}, \textit{Car} \rangle \\ \text{associates(Employment)} &= \langle \textit{Branch}, \textit{Employee} \rangle \\ \text{associates(Fleet)} &= \langle \textit{Branch}, \textit{Car} \rangle \\ \text{associates(Maintenance)} &= \langle \textit{ServiceDepot}, \textit{Check}, \textit{Car} \rangle \\ \text{associates(Management)} &= \langle \textit{Branch}, \textit{Employee} \rangle \\ \text{associates(Offer)} &= \langle \textit{Branch}, \textit{CarGroup} \rangle \\ \text{associates(Provider)} &= \langle \textit{Rental}, \textit{Branch} \rangle \\ \text{associates(Quality)} &= \langle \textit{CarGroup}, \textit{CarGroup} \rangle \\ \text{associates(Reservation)} &= \langle \textit{Rental}, \textit{CarGroup} \rangle \end{aligned}$$

□

Role names

Classes may appear more than once in an association each time playing a different role. For example, in a self-association PhoneCall on a class *Person*

we need to distinguish between the person having the role of a caller and another person being the callee. Therefore we assign each class participating in an association a unique role name. Role names are also important for OCL navigation expressions. A role name of a class is used to determine the navigation path in this kind of expressions.

Definition 3.5 (Role names)
Let $as \in \text{ASSOC}$ be an association with associates$(as) = \langle c_1, \ldots, c_n \rangle$. Role names for an association are defined by a function

$$\text{roles} : \begin{cases} \text{ASSOC} \to \mathcal{N}^+ \\ as \mapsto \langle r_1, \ldots, r_n \rangle \text{ with } (n \geq 2) \end{cases}$$

where all role names must be distinct, i.e.,

$$\forall i, j \in \{1, \ldots, n\} : i \neq j \implies r_i \neq r_j \ .$$

□

The function roles$(as) = \langle r_1, \ldots, r_n \rangle$ assigns each class c_i for $1 \leq i \leq n$ participating in the association a unique role name r_i. If role names are omitted in a class diagram, implicit names are constructed in UML by using the name of the class at the target end and changing its first letter to lower case. For example, the role name of the association Maintenance at the association end targeting the class *ServiceDepot* is "serviceDepot". As mentioned above, explicit role names are mandatory for self-associations. For example, the self-association Quality has role names lower and higher.

Example. These are the role names of classes participating in associations in our example.

$$\begin{aligned}
&\text{roles(Assignment)} &&= \langle \text{rental}, \text{car} \rangle \\
&\text{roles(Booking)} &&= \langle \text{rental}, \text{customer} \rangle \\
&\text{roles(Classification)} &&= \langle \text{carGroup}, \text{car} \rangle \\
&\text{roles(Employment)} &&= \langle \text{employer}, \text{employee} \rangle \\
&\text{roles(Fleet)} &&= \langle \text{branch}, \text{car} \rangle \\
&\text{roles(Maintenance)} &&= \langle \text{serviceDepot}, \text{check}, \text{car} \rangle \\
&\text{roles(Management)} &&= \langle \text{managedBranch}, \text{manager} \rangle \\
&\text{roles(Offer)} &&= \langle \text{branch}, \text{carGroup} \rangle \\
&\text{roles(Provider)} &&= \langle \text{rental}, \text{branch} \rangle \\
&\text{roles(Quality)} &&= \langle \text{lower}, \text{higher} \rangle \\
&\text{roles(Reservation)} &&= \langle \text{rental}, \text{carGroup} \rangle
\end{aligned}$$

□

Additional syntactical constraints are required for ensuring the uniqueness of role names when a class is part of many associations. We first define a

function participating that gives the set of associations a class participates in.

$$\text{participating} : \begin{cases} \textsc{Class} \to \mathcal{P}(\textsc{Assoc}) \\ c \mapsto \{as \mid as \in \textsc{Assoc} \wedge \text{associates}(as) = \langle c_1, \ldots, c_n \rangle \\ \qquad\qquad \wedge \exists i \in \{1, \ldots, n\} : c_i = c\} \end{cases}$$

Example. The class *Car* participates in four associations.

$$\text{participating}(\mathit{Car}) = \{\text{Assignment}, \text{Classification}, \text{Fleet}, \text{Maintenance}\}$$

□

The following function navends gives the set of all role names reachable (or *navigable*) from a class over a given association.

$$\text{navends} : \begin{cases} \textsc{Class} \times \textsc{Assoc} \to \mathcal{P}(\mathcal{N}) \\ (c, as) \mapsto \{r \mid \text{associates}(as) = \langle c_1, \ldots, c_n \rangle \\ \qquad\qquad \wedge \text{roles}(as) = \langle r_1, \ldots, r_n \rangle \\ \qquad\qquad \wedge \exists i, j \in \{1, \ldots, n\} : (i \neq j \wedge c_i = c \wedge r_j = r)\} \end{cases}$$

Example. Role names reachable from class *Car* and *CarGroup* for the associations Maintenance, Classification, and Quality are given. Note that for the self-association Quality both role names are reachable from *CarGroup*.

$$\begin{aligned} \text{navends}(\mathit{Car}, \text{Maintenance}) &= \{\text{check}, \text{serviceDepot}\} \\ \text{navends}(\mathit{Car}, \text{Classification}) &= \{\text{carGroup}\} \\ \text{navends}(\mathit{CarGroup}, \text{Quality}) &= \{\text{higher}, \text{lower}\} \end{aligned}$$

□

With this function, we can easily express a well-formedness rule specifying that role names that are reachable from a class over *all* associations the class participates in must be distinct.

$$\forall as_1, as_2 \in \text{participating}(c) : \\ (as_1 \neq as_2 \implies \text{navends}(c, as_1) \cap \text{navends}(c, as_2) = \emptyset) \quad (\textbf{WF-1})$$

The set of role names that are reachable from a class along all associations the class participates in can then be determined by the following function.

$$\text{navends}(c) : \begin{cases} \textsc{Class} \to \mathcal{P}(\mathcal{N}) \\ c \mapsto \bigcup_{as \in \text{participating}(c)} \text{navends}(c, as) \end{cases}$$

Example. These are all role names reachable from class *Car* over all associations the class participates in (Assignment, Classification, Fleet, and Maintenance).

$$\text{navends}(\mathit{Car}) = \{\text{check}, \text{serviceDepot}, \text{rental}, \text{carGroup}, \text{branch}\}$$

□

Multiplicities

An association specifies the possible existence of links between objects of associated classes. The number of links that an object can be part of is specified with *multiplicities*. This concept is also known as cardinality ratio in Entity-Relationship modeling. There are a number of different notation styles and different levels of detail for cardinality ratios in the various ER dialects. A multiplicity specification in UML can be represented by a set of natural numbers.

Definition 3.6 (Multiplicities)
Let $as \in \textsc{Assoc}$ be an association with associates$(as) = \langle c_1, \ldots, c_n \rangle$. The function multiplicities$(as) = \langle M_1, \ldots, M_n \rangle$ assigns each class c_i participating in the association a non-empty set $M_i \subseteq \mathbb{N}_0$ with $M_i \neq \{0\}$ for all $1 \leq i \leq n$.
□

For example, the multiplicities of the Maintenance association are defined as

$$\text{multiplicities}(\text{Maintenance}) = \langle \{0,1\}, \mathbb{N}_0, \mathbb{N}_0 \rangle.$$

The precise meaning of multiplicities is defined as part of the interpretation of object models in Section 3.4.

Remark: aggregation and composition

Special forms of associations are aggregation and composition. In general, aggregations and compositions impose additional restrictions on relationships. An aggregation is a special kind of binary association representing a *part-of* relationship. The aggregate is marked with a hollow diamond at the association end in class diagrams. An aggregation implies the constraint that an object cannot be part of itself. Therefore, a link of an aggregation may not connect the same object. In case of chained aggregations, the chain may not contain cycles.

An even stronger form of aggregation is composition. The composite is marked with a filled diamond at the association end in class diagrams. In

addition to the requirements for aggregations, a part may only belong to at most one composite.

These seemingly simple concepts can have quite complex semantic issues [AFGP96, Mot96, Pri97, GR99, HSB99, BHS99, BHSOG01]. Here, we are concerned only with syntax. The syntax of aggregations and compositions is very similar to associations. Therefore, we do not add an extra concept to our formalism. As a convention, we always use the first component in an association for a class playing the role of an aggregate or composite. The semantic restrictions then have to be expressed as an explicit constraint. A systematic way for mapping aggregations and compositions to simple associations plus OCL constraints is presented in [GR99].

3.3.6 Generalization

A generalization is a taxonomic relationship between two classes. This relationship specializes a general class into a more specific class. Specialization and generalization are different views of the same concept. Generalization relationships form a hierarchy over the set of classes.

Definition 3.7 (Generalization hierarchy)
A generalization hierarchy $\prec$ is a partial order on the set of classes $\textsc{Class}$. □

Pairs in $\prec$ describe generalization relationships between two classes. For classes $c_1, c_2 \in \textsc{Class}$ with $c_1 \prec c_2$, the class c_1 is called a *child class* of c_2, and c_2 is called a *parent class* of c_1.

Example. The class diagram shown in Figure 2.2 contains two generalizations. *Person* is the parent class of *Customer* and *Employee*.

$$\prec = \{(\mathit{Customer}, \mathit{Person}), (\mathit{Employee}, \mathit{Person})\}$$

□

Full descriptor of a class

A child class implicitly inherits attributes, operations and associations of its parent classes. The set of properties defined in a class together with its inherited properties is called a *full descriptor* in UML [OMG99e, p 2-60]. We can formalize the full descriptor in our framework as follows. First, we define a convenience function for collecting all parents of a given class.

$$\text{parents} : \begin{cases} \textsc{Class} \to \mathcal{P}(\textsc{Class}) \\ c \mapsto \{c' \mid c' \in \textsc{Class} \wedge c \prec c'\} \end{cases}$$

The full set of attributes of class c is the set ATT_c^* containing all inherited attributes and those that are defined directly in the class.

$$\text{ATT}_c^* = \text{ATT}_c \cup \bigcup_{c' \in \text{parents}(c)} \text{ATT}_{c'}$$

We define the set of inherited user-defined operations analogously.

$$\text{OP}_c^* = \text{OP}_c \cup \bigcup_{c' \in \text{parents}(c)} \text{OP}_{c'}$$

Finally, the set of navigable role names for a class and all of its parents is given as follows.

$$\text{navends}^*(c) = \text{navends}(c) \cup \bigcup_{c' \in \text{parents}(c)} \text{navends}(c')$$

Definition 3.8 (Full descriptor of a class)
The full descriptor of a class $c \in \text{CLASS}$ is a structure $\text{FD}_c = (\text{ATT}_c^*, \text{OP}_c^*, \text{navends}^*(c))$ containing all attributes, user-defined operations, and navigable role names defined for the class and all of its parents. □

The UML standard requires that properties of a full descriptor must be distinct. For example, a class may not define an attribute that is already defined in one of its parent classes. These constraints are captured more precisely be the following well-formedness rules in our framework. Each constraint must hold for each class $c \in \text{CLASS}$.

1. Attributes are defined in exactly one class.

$$\forall (a : t_c \to t,\ a' : t_{c'} \to t' \in \text{ATT}_c^*) : \\ (a = a' \implies t_c = t_{c'} \wedge t = t') \qquad \textbf{(WF-2)}$$

2. In a full class descriptor, an operation may only be defined once. The first parameter of an operation signature indicates the class in which the operation is defined. The following condition guarantees that each operation in a full class descriptor is defined in a single class.

$$\forall (\omega : t_c \times t_1 \times \cdots \times t_n \to t,\ \omega : t_{c'} \times t_1 \times \cdots \times t_n \to t \in \text{OP}_c^*) : \\ (c = c') \qquad \textbf{(WF-3)}$$

   An operation of a class is called *overloaded* if there are one or more operations in the full descriptor with the same symbol but different number or type of parameters. A closely related notion that should not

be confused with overloading is that of *overriding*. A method implementing an operation may be overridden by another method in a child class that implements the same operation thus effectively replacing the method of the parent. Since we are focusing on the specification level, we do not need to consider here the implementation of operations with methods.

3. Role names are defined in exactly one class.

$$\forall c_1, c_2 \in \text{parents}(c) \cup \{c\} : \\ (c_1 \neq c_2 \implies \text{navends}(c_1) \cap \text{navends}(c_2) = \emptyset) \qquad \textbf{(WF-4)}$$

4. Attribute names and role names must not conflict. This is necessary because in OCL the same notation is used for attribute access and navigation by role name. For example, the expression `self.x` may either be a reference to an attribute `x` or a reference to a role name `x`.

$$\forall (a : t_c \to t \in \text{ATT}_c^*) : \forall r \in \text{navends}^*(c) : \\ (a \neq r) \qquad \textbf{(WF-5)}$$

Note that operations may have the same name as attributes or role names because the concrete syntax of OCL allows us to distinguish between these cases. For example, the expression `self.age` is either an attribute or role name reference, but a call to an operation age without parameters is written as `self.age()`.

3.3.7 Formal Syntax

We combine the components introduced in the previous section to formally define the syntax of object models.

Definition 3.9 (Syntax of object models)
The syntax of an object model is a structure

$$\mathcal{M} = (\text{CLASS}, \text{ATT}_c, \text{OP}_c, \text{ASSOC}, \text{associates}, \text{roles}, \text{multiplicities}, \prec)$$

where

i. CLASS is a set of classes (Definition 3.1).

ii. ATT_c is a set of operation signatures for functions mapping an object of class c to an associated attribute value (Definition 3.2).

iii. OP_c is a set of signatures for user-defined operations of a class c (Definition 3.3).

iv. ASSOC is a set of association names (Definition 3.4).

   (a) associates is a function mapping each association name to a list of participating classes (Definition 3.4).
   (b) roles is a function assigning each end of an association a role name (Definition 3.5).
   (c) multiplicities is a function assigning each end of an association a multiplicity specification (Definition 3.6).

v. $\prec$ is a partial order on CLASS reflecting the generalization hierarchy of classes (Definitions 3.7 and 3.8).

□

3.4 Interpretation of Object Models

In the previous section, the syntax of object models has been defined. An interpretation of object models is presented in the following.

3.4.1 Objects

The domain of a class $c \in \text{CLASS}$ is the set of objects that can be created by this class and all of its child classes. Objects are referred to by unique object identifiers. In the following, we will make no conceptual distinction between objects and their identifiers. Each object is uniquely determined by its identifier and vice versa. Therefore, the actual representation of an object is not important for our purposes.

Definition 3.10 (Object identifiers)

i. The set of object identifiers of a class $c \in \text{CLASS}$ is defined by an infinite set $\text{oid}(c) = \{\underline{c}_1, \underline{c}_2, \dots\}$.

ii. The domain of a class $c \in \text{CLASS}$ is defined as $I_{\text{CLASS}}(c) = \bigcup\{\text{oid}(c') \mid c' \in \text{CLASS} \wedge c' \preceq c\}$.

□

In the following, we will omit the index for a mapping I when the context is obvious. The concrete scheme for naming objects is not important as long as every object can be uniquely identified, i.e., there are no different objects having the same name. We sometimes use single letters combined with increasing indexes to name objects if it is clear from the context to which class these objects belong.

Example. Semantic domains for some classes of our example model are given. Note that the domain of class *Person* includes the domains of its child classes *Employee* and *Customer*.

$$\begin{aligned} I(Branch) &= \{\underline{Branch}_1, \underline{Branch}_2, \dots\} \\ I(Customer) &= \{\underline{cu}_1, \underline{cu}_2, \dots\} \\ I(Employee) &= \{\underline{e}_1, \underline{e}_2, \dots\} \\ I(Person) &= \{\underline{p}_1, \underline{p}_2, \dots\} \cup I(Customer) \cup I(Employee) \end{aligned}$$

□

Generalization

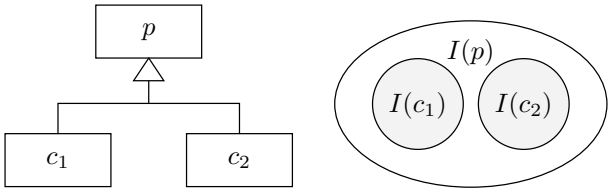
The above definition implies that a generalization hierarchy induces a subset relation on the semantic domain of classes. The set of object identifiers of a child class is a subset of the set of object identifiers of its parent classes. With other words, we have

$$\forall c_1, c_2 \in \textsc{Class} : c_1 \prec c_2 \implies I(c_1) \subseteq I(c_2) \ .$$

From the perspective of programming languages this closely corresponds to the domain-inclusion semantics commonly associated with subtyping and inheritance [CW85]. Data models for object-oriented databases such as the generic OODB model presented in [AHV95] also assume an inclusion semantics for class extensions. This requirement guarantees two fundamental properties of generalizations. First, an object of a child class has (inherits) all the properties of its parent classes because it *is* an instance of the parent classes. Second, this implies that an object of a more specialized class can be used anywhere where an object of a more general class is expected (principle of substitutability) because it has at least all the properties of the parent classes.

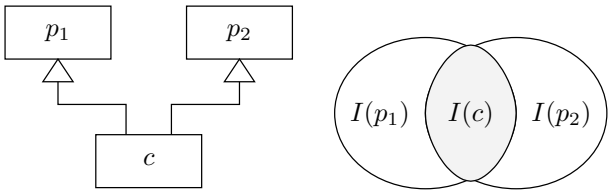
In general, the interpretation of classes is pairwise disjoint if two classifiers are not related by generalization and do not have a common child. Figure 3.1 illustrates this situation. It shows the case of single inheritance where a class does not have more than one parent. Classes c_1 and c_2 are children of the class p. The interpretations $I(c_1)$ and $I(c_2)$ are subsets of $I(p)$. These subsets are disjoint because c_1 and c_2 are not related to each other by means of generalization.

Figure 3.2 shows an example of multiple parent classes (also called multiple inheritance in object-oriented languages). Objects of the child class c are instances of the parent classes p_1 and p_2 at the same time.



(a) Class diagram **(b)** Interpretation

Figure 3.1: Single inheritance and its interpretation



(a) Class diagram **(b)** Interpretation

Figure 3.2: Multiple inheritance and its interpretation

In general, only the domains of leaf classes are disjoint. The concept of disjoint object identifiers makes it easier to associate objects with classes. An object identifier uniquely identifies the class to which an object belongs [AHV95].

3.4.2 Links

An association describes possible connections between objects of the classes participating in the association. A connection is also called a link in UML terminology.

Definition 3.11 (Links)
Each association $as \in \text{ASSOC}$ with $\text{associates}(as) = \langle c_1, \ldots, c_n \rangle$ is interpreted as the Cartesian product of the sets of object identifiers of the participating classes: $I_{\text{ASSOC}}(as) = I_{\text{CLASS}}(c_1) \times \cdots \times I_{\text{CLASS}}(c_n)$. A *link* denoting a connection between objects is an element $l_{as} \in I_{\text{ASSOC}}(as)$. □

The interpretation of an association is a relation describing the set of all possible links between objects of the associated classes and their children.

Example. The interpretation of the association Assignment is a binary relation whereas the association Maintenance is interpreted by a ternary relation. The self-association Quality describes links between *CarGroup* objects.

$$I(\text{Assignment}) = I(\mathit{Rental}) \times I(\mathit{Car})$$
$$I(\text{Maintenance}) = I(\mathit{ServiceDepot}) \times I(\mathit{Check}) \times I(\mathit{Car})$$
$$I(\text{Quality}) = I(\mathit{CarGroup}) \times I(\mathit{CarGroup})$$

□

3.4.3 System State

Objects, links and attribute values constitute the state of a system at a particular moment in time. A system is in different states as it changes over time. Therefore, a system state is also called a snapshot of a running system. With respect to OCL, we can in many cases concentrate on a single system state given at a discrete point in time. For example, a system state provides the complete context for the evaluation of OCL invariants. For pre- and postconditions, however, it is necessary to consider two consecutive states.

Definition 3.12 (System state)
A system state for a model $\mathcal{M}$ is a structure $\sigma(\mathcal{M}) = (\sigma_{\text{CLASS}}, \sigma_{\text{ATT}}, \sigma_{\text{ASSOC}})$.

i. The finite sets $\sigma_{\text{CLASS}}(c)$ contain all objects of a class $c \in \text{CLASS}$ existing in the system state: $\sigma_{\text{CLASS}}(c) \subset \text{oid}(c)$.

ii. Functions σ_{ATT} assign attribute values to each object: $\sigma_{\text{ATT}}(a) : \sigma_{\text{CLASS}}(c) \to I(t)$ for each $a : t_c \to t \in \text{ATT}_c^*$.

iii. The finite sets σ_{ASSOC} contain links connecting objects. For each $as \in \text{ASSOC}$: $\sigma_{\text{ASSOC}}(as) \subset I_{\text{ASSOC}}(as)$. A link set must satisfy all multiplicity specifications defined for an association (the function $\pi_i(l)$ projects the ith component of a tuple or list l, whereas the function $\bar{\pi}_i(l)$ projects *all but* the ith component):

$$\forall i \in \{1, \ldots, n\}, \forall l \in \sigma_{\text{ASSOC}}(as) :$$
$$|\{l' \mid l' \in \sigma_{\text{ASSOC}}(as) \wedge (\bar{\pi}_i(l') = \bar{\pi}_i(l))\}| \in \pi_i(\text{multiplicities}(as))$$

□

Example. Figure 3.3 shows a small, yet complete, system state of the car rental model as a UML object diagram. The system state includes one *Branch* object and two *Employee* objects. Both employee objects are

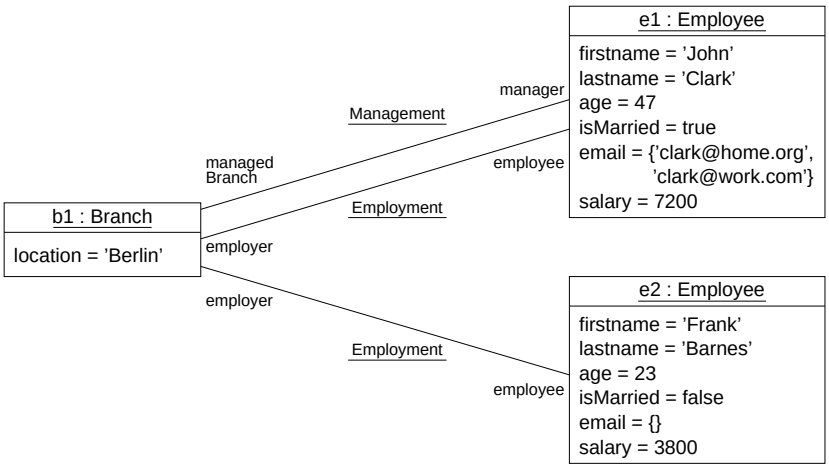


Figure 3.3: Object diagram showing a system state of the car rental model

connected to the branch object by links of the Employment association. A link of the Management association shows that the employee denoted by object $\underline{e}_1$ is also the manager of the branch.

The information in the object diagram can be formally described as a state $\sigma(\text{CarRental}) = (\sigma_{\text{CLASS}}, \sigma_{\text{ATT}}, \sigma_{\text{ASSOC}})$ as follows.

$$
\begin{aligned}
&\sigma_{\text{CLASS}}(\mathit{Branch}) = \{\underline{b}_1\}\\
&\sigma_{\text{CLASS}}(\mathit{Employee}) = \{\underline{e}_1, \underline{e}_2\}\\
&\sigma_{\text{ATT}}(\text{location})(\underline{b}_1) = \text{'Berlin'}\\
&\sigma_{\text{ATT}}(\text{firstname})(\underline{e}_1) = \text{'John'}\\
&\sigma_{\text{ATT}}(\text{lastname})(\underline{e}_1) = \text{'Clark'}\\
&\sigma_{\text{ATT}}(\text{age})(\underline{e}_1) = 47\\
&\sigma_{\text{ATT}}(\text{isMarried})(\underline{e}_1) = \text{true}\\
&\sigma_{\text{ATT}}(\text{email})(\underline{e}_1) = \{\text{'clark@home.org'}, \text{'clark@work.com'}\}\\
&\sigma_{\text{ATT}}(\text{salary})(\underline{e}_1) = 7200\\
&\sigma_{\text{ATT}}(\text{firstname})(\underline{e}_2) = \text{'Frank'}\\
&\sigma_{\text{ATT}}(\text{lastname})(\underline{e}_2) = \text{'Barnes'}\\
&\sigma_{\text{ATT}}(\text{age})(\underline{e}_2) = 23\\
&\sigma_{\text{ATT}}(\text{isMarried})(\underline{e}_2) = \text{false}\\
&\sigma_{\text{ATT}}(\text{email})(\underline{e}_2) = \{\}\\
&\sigma_{\text{ATT}}(\text{salary})(\underline{e}_2) = 3800\\
&\sigma_{\text{ASSOC}}(\text{Management}) = \{(\underline{b}_1, \underline{e}_1)\}\\
&\sigma_{\text{ASSOC}}(\text{Employment}) = \{(\underline{b}_1, \underline{e}_1), (\underline{b}_1, \underline{e}_2)\}
\end{aligned}
$$

All other sets being part of $\sigma(\text{CarRental})$ that are not mentioned above are empty in this example, e.g., $\sigma_{\text{CLASS}}(\mathit{Car}) = \emptyset$. Note that the set of attribute

values of an object is determined by the full descriptor of its class. All attribute values of an employee object except for the salary are inherited from class *Person*. □

Our definition of system state is adequate for model features defined in Section 3.3. Additional constraints would be necessary if aggregation and composition were added to the model. The most important requirement is that no object can be – directly or indirectly – part of itself. Thus, the transitive closure of the sets σ_{Assoc} must be irreflexive for binary associations. We avoid this complexity here, because this and other constraints related to properties of aggregation and composition can be expressed by adding equivalent OCL constraints to a model that uses only simple associations [GR99].

An alternative approach to defining system states is given in [RG98]. There we used a hypergraph approach which facilitates a direct and intuitive mapping of a state to an object diagram. The nodes of the graph are given by the set of objects. Nodes are labeled with attribute values, and edges are links between objects. The present equivalent approach has been chosen because it is slightly easier to define the semantics of navigation operations (see Section 4.4).

3.4.4 Formal Interpretation of Object Models

The semantics of an object model is the set of all possible system states.

Definition 3.13 (Interpretation of object models)
The interpretation of an object model $\mathcal{M}$ is the set of all possible system states $\sigma(\mathcal{M})$. □

3.5 UML Model of BML Concepts

The previous sections presented all basic concepts necessary to specify object models. Furthermore, the given semantics assigns each concept a precise meaning. Although the formalism established in this chapter helps to be more precise, it also adds some complexity. Therefore, we present an alternative view in Figure 3.4 highlighting the main concepts.

The diagram shows fundamental concepts and their relationships by using UML classes and associations. The left part of the diagram shows elements of the syntax of object models (Model, Attribute, Class, Association, Generalization), while the right part shows elements being part of the interpretation of object models (AttrValue, Object, Link, SystemState).

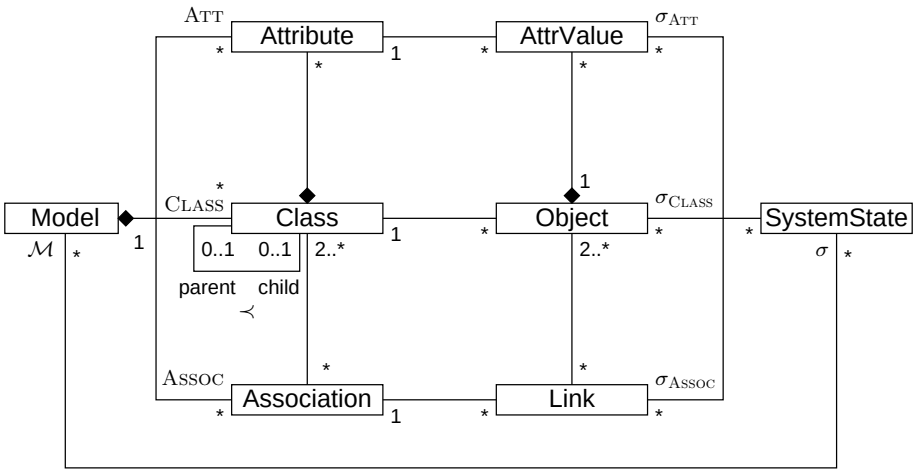


Figure 3.4: Main concepts of object models as a UML class diagram

The model shown in Figure 3.4 illustrates fundamental structures of UML. The left part of the diagram is a simplified view of the Core package of the UML metamodel [OMG99e]. The right part shows central elements of the Common Behavior package of the metamodel. A major difference is that UML does not have the notion of system state. As a consequence, it seems impossible with UML to model an object in different states having, for example, different attribute values.

3.6 Discussion

The object model presented in this chapter is similar to object-oriented data models known from conceptual modeling. In the following, we compare our approach with the *Extended Entity-Relationship* (EER) approach [Gog94] for schema definitions.

- EER schemata do not contain user-defined operations because the focus lies on modeling static structure. We include operations because they are the target of OCL specifications of pre- and postconditions.
- The EER approach provides a more powerful *type construction* mechanism for modeling various aspects of generalizations.
- In the EER approach, entity types can be directly used as sorts. The interpretation includes an undefined value for each entity type. In our approach, we separate between classes and types. We need to consider undefined values only for types, whereas the interpretation of classes

exclusively consists of valid objects. The former is useful with respect to OCL expressions, the latter seems more appropriate for describing the object model.

- Part-of relationships are included in the EER approach. We only provide simple associations, because aggregations and compositions can be treated equivalently in OCL, thus simplifying the object model.

Summary

In this chapter, we formally defined object models containing classes, attributes, operations, generalization hierarchies, and associations. These basic concepts constitute fundamental modeling elements in UML for describing static structural aspects of a system. The syntax of each of these concepts has been defined. A precise semantics has been given by an interpretation of object models that maps elements of the syntax to a semantic domain. System states representing snapshots of a system containing objects, links, and attribute values have been introduced. A system state can be visualized as a UML object diagram. The formalism defined in this chapter provides the necessary framework for a detailed discussion of OCL in the following chapters.

Chapter 4

OCL Types and Operations

OCL is a strongly typed language. A type is assigned to every OCL expression and typing rules determine in which ways well-formed expressions can be constructed. In addition to those types introduced by UML models, there are a number of predefined OCL types and operations available for use with any UML model. This chapter formally defines the type system of OCL. Types and their domains are fixed, and the abstract syntax and semantics of operations is defined.

Section 4.1 gives a brief overview of the OCL type system. Section 4.2 defines the basic types *Integer*, *Real*, *Boolean* and *String*. Enumeration types are defined in Section 4.3. Section 4.4 introduces object types that correspond to classes in a model. Collection types are discussed in Section 4.5. Special types such as *OclAny* and *OclType* are considered in Section 4.6. Section 4.7 introduces subtype relationships forming a type hierarchy. All types and operations are finally summarized in a data signature defined in Section 4.8. In the final section, we discuss possible extensions to the type system for addressing advanced requirements.

4.1 Concepts

An overview of the types defined in this chapter is given in Figure 4.1. The diagram shows the names of types and the subtype relationship among different types. An arrow from one type to another indicates that the former is a subtype of the latter. The subtype relation is transitive, for example, *Integer* is a subtype of *Real*, and both are subtypes of *OclAny*.

Types in OCL can be classified as follows. The group of predefined basic types includes *Integer*, *Real*, *Boolean*, and *String*. Enumeration types are

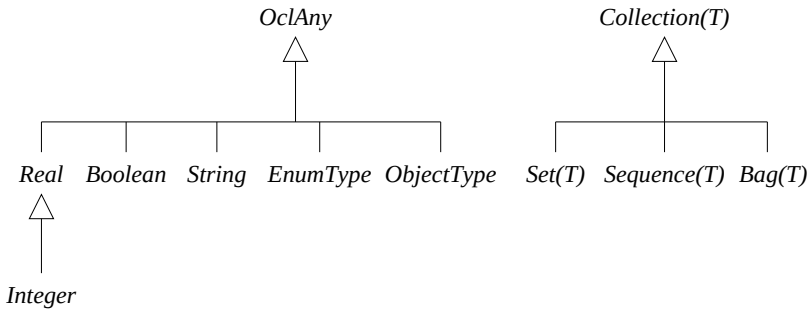


Figure 4.1: Overview of OCL types

user-defined. An object type corresponds to a classifier in an object model. The supertype of all these types is *OclAny*.

Collections of values can be described by the collection types $Set(T)$, $Sequence(T)$, and $Bag(T)$. These are the classical types for bulk data, namely sets, lists, and multi-sets, respectively. The parameter T denotes the type of the elements. A common supertype of the collection types is $Collection(T)$.

Figure 4.1 is based on information in [OMG99b]. It contains almost all important types available in OCL. However, some types such as *OclExpression* are left out because there is very little explanation and motivation in the standard documents. These issues are discussed in-depth in the relevant sections throughout this chapter.

Our general approach to defining the type system is as follows. Types are associated with a set of operations. These operations describe functions combining or operating on values of the type domains. In our approach, we use a data signature $\Sigma = (T, \Omega)$ to describe the syntax of types and operations. The semantics of types in T and operations in Ω is defined by a mapping that assigns each type a domain and each operation a function. The definition of the syntax and semantics of types and operations will be developed and extended in several steps. At the end of this chapter, the complete set of types is defined in a single data signature.

4.2 Basic Types

Basic types are *Integer*, *Real*, *Boolean* and *String*. The syntax of basic types and their operations is defined by a signature $\Sigma_B = (T_B, \Omega_B)$. T_B is the set of basic types, Ω_B is the set of signatures describing operations over basic types.

Definition 4.1 (Syntax of basic types)
The set of basic types T_B is defined as $T_B = \{Integer, Real, Boolean, String\}$.
□

Next we define the semantics of basic types by mapping each type to a domain.

Definition 4.2 (Semantics of basic types)
Let $\mathcal{A}^*$ be the set of finite sequences of characters from a finite alphabet $\mathcal{A}$. The semantics of a basic type $t \in T_B$ is a function I mapping each type to a set:

- $I(Integer) = \mathbb{Z} \cup \{\perp\}$
- $I(Real) = \mathbb{R} \cup \{\perp\}$
- $I(Boolean) = \{\text{true}, \text{false}\} \cup \{\perp\}$
- $I(String) = \mathcal{A}^* \cup \{\perp\}$.

□

The basic type *Integer* represents the set of integers, *Real* the set of real numbers, *Boolean* the truth values true and false, and *String* all finite strings over a given alphabet. Each domain also contains a special undefined value which is motivated in the next section.

4.2.1 Error Handling

Each domain of a basic type t contains a special value $\perp$. This value represents an undefined value which is useful for two purposes.

1. An undefined value may, for example, be assigned to an attribute of an object. In this case the undefined value helps to model the situation where the attribute value is not yet known (for example, the email address of a customer is unknown at the time of the first contact, but will be added later) or does not apply to this specific object instance (e.g., the customer does not have an email address). This usage of undefined values is well-known in database modeling and querying with SQL [Dat90, EN94]), in the Extended ER-Model [Gog94], and in the object specification language TROLL *light* [Her95].

2. An undefined value can signal an error in the evaluation of an expression. An example for an expression that is defined by a partial function is the division of integers. The result of a division by zero is undefined. The problems with partial functions can be eliminated by including an undefined value $\perp$ into the domains of types. For all operations we can then extend their interpretation to total functions.

The interpretation of operations is considered strict unless there is an explicit statement in the following. Hence, an undefined argument value causes an undefined operation result. This ensures the propagation of error conditions.

4.2.2 Operations

There are a number of predefined operations on basic types. The set Ω_B contains the signatures of these operations. An operation signature describes the name, the parameter types, and the result type of an operation.

Definition 4.3 (Syntax of operations)
The syntax of an operation is defined by a signature $\omega : t_1 \times \cdots \times t_n \to t$. The signature contains the operation symbol ω, a list of parameter types $t_1, \ldots, t_n \in T$, and a result type $t \in T$. □

Table 4.1 shows a schema defining most predefined operations over basic types. The left column contains partially parameterized signatures in Ω_B. The right column specifies variations for the operation symbols or types in the left column.

The set of predefined operations includes the usual arithmetic operations $+, -, *, /$, etc. for integers and real numbers, division (div) and modulo (mod) of integers, sign manipulation ($-$, abs)[1], conversion of *Real* values to *Integer* values (floor, round), and comparison operations ($<, >, \leq, \geq$). Unfortunately, the comparison operations are not defined for *String* and *Boolean* values in OCL. We include them here for reasons of orthogonality and completeness.

Operations for equality and inequality are presented later in this section, since they apply to all types. Boolean values can be combined in different ways (and, or, xor, implies), and they can be negated (not). For strings the length of a string (size) can be determined, a string can be projected to a substring and two strings can be concatenated (concat). Finally, assuming a standard alphabet like ASCII or Unicode, case translations are possible with toUpper and toLower.

[1] The unary operator $-$ only appears in a production of the OCL grammar [OMG99b], not in the list of standard operations.

Signature	Schema parameters
$\omega : \mathit{Integer} \times \mathit{Integer} \to \mathit{Integer}$	$\omega \in \{+, -, *, \max, \min\}$
$\mathit{Integer} \times \mathit{Real} \to \mathit{Real}$	
$\mathit{Real} \times \mathit{Integer} \to \mathit{Real}$	
$\mathit{Real} \times \mathit{Real} \to \mathit{Real}$	
$\omega : \mathit{Integer} \times \mathit{Integer} \to \mathit{Integer}$	$\omega \in \{\mathrm{div}, \mathrm{mod}\}$
$/ : t_1 \times t_2 \to \mathit{Real}$	$t_1, t_2 \in \{\mathit{Integer}, \mathit{Real}\}$
$- : t \to t$	$t \in \{\mathit{Integer}, \mathit{Real}\}$
$\mathrm{abs} : t \to t$	
$\mathrm{floor} : t \to \mathit{Integer}$	
$\mathrm{round} : t \to \mathit{Integer}$	
$\omega : t_1 \times t_2 \to \mathit{Boolean}$	$\omega \in \{<, >, \leq, \geq\}$, $t_1, t_2 \in \{\mathit{Integer}, \mathit{Real}, \mathit{String}, \mathit{Boolean}\}$
$\omega : \mathit{Boolean} \times \mathit{Boolean} \to \mathit{Boolean}$	$\omega \in \{\mathrm{and}, \mathrm{or}, \mathrm{xor}, \mathrm{implies}\}$
$\mathrm{not} : \mathit{Boolean} \to \mathit{Boolean}$	
$\mathrm{size} : \mathit{String} \to \mathit{Integer}$	
$\mathrm{concat} : \mathit{String} \times \mathit{String} \to \mathit{String}$	
$\mathrm{toUpper} : \mathit{String} \to \mathit{String}$	
$\mathrm{toLower} : \mathit{String} \to \mathit{String}$	
$\mathrm{substring} : \mathit{String} \times \mathit{Integer} \times \mathit{Integer} \to \mathit{String}$	

Table 4.1: Schema for operations on basic types

Some operation symbols (such as + and −) are overloaded, that is there are signatures having the same operation symbol but different parameters (concerning number or type) and possibly different result types. Thus in general, the full argument list has to be considered in order to identify a signature unambiguously.

The operations in Table 4.1 all have at least one parameter. There is another set of operations in Ω_B which do not have parameters. These operations are used to produce constant values of basic types. For example, the integer value 12 can be generated by the operation $12 :\to \mathit{Integer}$. Similar operations exist for the other basic types. For each value, there is an operation with no parameters and an operation symbol that corresponds to the common notational representation of this value.

4.2.3 Semantics of Operations

Definition 4.4 (Semantics of operations)
The semantics of an operation with signature $\omega : t_1 \times \cdots \times t_n \to t$ is a total function $I(\omega : t_1 \times \cdots \times t_n \to t) : I(t_1) \times \cdots \times I(t_n) \to I(t)$. □

When we refer to an operation, we usually omit the specification of the parameter and result types and only use the operation symbol if the full signature can be derived from the context.

The next example shows the interpretation of the operation + for adding two integers. The operation has two arguments $i_1, i_2 \in I(Integer)$. This example also demonstrates the strict evaluation semantics for undefined arguments.

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

We can define the semantics of the other operations in Table 4.1 analogously. The usual semantics of the boolean operations and, or, xor, implies, and not, is extended for dealing with undefined argument values. Table 4.2 shows the interpretation of boolean operations following the proposal in [CKM+99a] based on three-valued logic.

b_1	b_2	b_1 and b_2	b_1 or b_2	b_1 xor b_2	b_1 implies b_2	not b_1
false	false	false	false	false	true	true
false	true	false	true	true	true	true
true	false	false	true	true	false	false
true	true	true	true	false	true	false
false	$\bot$	false	$\bot$	$\bot$	true	true
true	$\bot$	$\bot$	true	$\bot$	$\bot$	false
$\bot$	false	false	$\bot$	$\bot$	$\bot$	$\bot$
$\bot$	true	$\bot$	true	$\bot$	true	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Table 4.2: Semantics of boolean operations

Since the semantics of the other basic operations for *Integer*, *Real*, and *String* values is rather obvious, we will not further elaborate on them here.

4.2.4 Common Operations on all Types

At this point, we introduce some operations that are defined on all types (including those which are defined in subsequent sections). For each type $t \in T$, the constant operation $\text{undefined}_t :\rightarrow t$ generates the undefined value $\bot$. The semantics is given by $I(\text{undefined}_t) = \bot$. The equality of values of the same type can be checked with the operation $=_t$: $t \times t \rightarrow Boolean$. Furthermore, the semantics of $=_t$ defines undefined values to be equal. For two values $v_1, v_2 \in I(t)$, we have

$$I(=_t)(v_1, v_2) = \begin{cases} \text{true} & \text{if } v_1 = v_2, \text{ or } v_1 = \bot \text{ and } v_2 = \bot, \\ \text{false} & \text{otherwise.} \end{cases}$$

A test for inequality $\neq_t$: $t \times t \to \mathit{Boolean}$ can be defined analogously. It is also useful to have an operation that allows to check whether an arbitrary value is well-defined or undefined. This can be done with the operations $\text{isDefined}_t : t \to \mathit{Boolean}$ and $\text{isUndefined}_t : t \to \mathit{Boolean}$.

$$I(\text{isDefined}_t)(v) = I(\neq)(v, \bot)$$
$$I(\text{isUndefined}_t)(v) = I(=)(v, \bot)$$

4.2.5 Discussion

OCL has a simple concept of undefined values. A sub-expression resulting in an undefined value makes the complete expression undefined [OMG99b, p. 7-11]. However, it is left open what kinds of expressions actually produce undefined values and how undefined values are interpreted, for example, when they are the result of a constraint.

The OCL rule of propagating undefined values basically corresponds to our strict evaluation semantics. In OCL there are two exceptions to the general rule concerning the boolean operations *and* and *or*. False and-ed with "anything" is false, true or-ed with "anything" is true. These exceptions not only are imprecise but appear to be incomplete with respect to the other boolean operations. For example, there are no explicit exception rules for the boolean operations implies, xor, and not. If we apply the general rule for undefined sub-expressions mentioned above, we get an inconsistency shown in Table 4.3. The equation (not b_1 or b_2) = (b_1 implies b_2) does not hold anymore. Our semantics shown in Table 4.2 ensures consistency.

b_1	b_2	not b_1	not b_1 or b_2	b_1 implies b_2
false	false	true	true	true
false	true	true	true	true
true	false	false	false	false
true	true	false	true	true
false	$\bot$	true	true	$\bot$
true	$\bot$	false	$\bot$	$\bot$
$\bot$	false	$\bot$	$\bot$	$\bot$
$\bot$	true	$\bot$	true	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Table 4.3: Inconsistency resulting from OCL rules for undefined values. The fourth and fifth column should yield equal results.

Problems with the concept of undefined values in OCL have first been mentioned in [GR98c]. There, we also considered the strict propagation rule of OCL to be too strong. For instance, we argued that $\bot = \bot$ should be true. The reason for this is that the addition of an undefined value to a

type domain should not change the general properties of the type. In this case, equality is an equivalence relation for each type. The OCL rule lets the whole expression be undefined since there are undefined sub-expressions. In our approach, the result will be true.

As another example, expressions that make explicit use of the undefined value are not possible with the general OCL rule. The expression "if true then 1 else $\perp$ endif" should result in 1 instead of $\perp$. This is the case with our definition of if-expressions in Chapter 5. We also discuss the behavior of universal and existential quantifiers together with undefined values in the following chapter.

To summarize, the concept of undefined values is important but unfortunately incomplete and apparently inconsistent in OCL. Since undefined values affect all types, operations and expressions, great care is necessary to integrate them consistently with the language.

4.3 Enumeration Types

Enumeration types are user-defined types. An enumeration type is defined by specifying a name and a set of literals. An enumeration value is one of the literals used for its type definition. Figure 4.2 shows how an enumeration type can be described in graphical UML notation [OMG99d, p. 3-54].

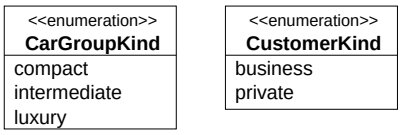


Figure 4.2: Enumeration types in graphical UML notation

The diagram reuses the classifier notation – a rectangle with compartments – for enumeration types. The first compartment defines the name of the type and has a stereotype ≪enumeration≫ to distinguish it from class definitions. The second compartment contains the literals.

The syntax of enumeration types and their operations is defined by a signature $\Sigma_E = (T_E, \Omega_E)$. T_E is the set of enumeration types and Ω_E the set of signatures describing the operations on enumeration types.

Definition 4.5 (Syntax of enumeration types)
An enumeration type $t \in T_E$ is associated with a finite non-empty set of enumeration literals by a function $\text{literals}(t) = \{e_{1_t}, \ldots, e_{n_t}\}$. □

An enumeration type is interpreted by the set of literals used for its declaration.

Definition 4.6 (Semantics of enumeration types)
The semantics of an enumeration type $t \in T_E$ is a function $I(t) = \text{literals}(t) \cup \{\bot\}$. □

The semantics of the enumeration types from the example in Figure 4.2 is defined as follows.

$$\begin{aligned} I(\mathit{CarGroupKind}) = \{&\text{compact}_{\text{CarGroupKind}}, \text{intermediate}_{\text{CarGroupKind}}, \\ &\text{luxury}_{\text{CarGroupKind}}, \bot\} \\ I(\mathit{CustomerKind}) = \{&\text{business}_{\text{CustomerKind}}, \text{private}_{\text{CustomerKind}}, \bot\} \end{aligned}$$

4.3.1 Operations

There is only a small number of operations defined on enumeration types: the test for equality or inequality of two enumeration values, a test for the undefined value, and the generation of an undefined enumeration value. The syntax and semantics of these general operations was defined in Section 4.2.4 on page 58 and applies to enumeration types as well.

4.3.2 Discussion

Enumeration types in OCL do not have a name. There is only a general type *Enumeration*, thus making a distinction between different enumerations impossible. For example, the equality operation is defined for the general OCL *Enumeration* type. This implies that values of different enumerations can be compared to each other and used wherever an enumeration is expected. This way different enumerations can be intermixed even when they are completely unrelated. Our definition avoids these problems.

4.4 Object Types

A central part of a UML model are classes that describe the structure of objects in a system. For each class, we define a corresponding object type describing the set of possible object instances. The syntax of object types and their operations is defined by a signature $\Sigma_C = (T_C, \Omega_C)$. T_C is the set of object types, and Ω_C is the set of signatures describing operations on object types.

Definition 4.7 (Syntax of object types)
Let $\mathcal{M}$ be a model with a set CLASS of class names. The set T_C of object types is defined such that for each class $c \in$ CLASS there is a type $t \in T_C$ having the same name as the class c. □

We define the following two functions for mapping a class to its type and vice versa.

$$\text{typeOf} : \text{CLASS} \rightarrow T_C$$
$$\text{classOf} : T_C \rightarrow \text{CLASS}$$

The interpretation of classes is used for defining the semantics of object types. The set of object identifiers $I_{\text{CLASS}}(c)$ was introduced in Definition 3.10 on page 44.

Definition 4.8 (Semantics of object types)
The semantics of an object type $t \in T_C$ with $\text{classOf}(t) = c$ is defined as $I(t) = I_{\text{CLASS}}(c) \cup \{\perp\}$. □

In summary, the domain of an object type is the set of object identifiers defined for the class and its children. The undefined value that is only available with the type – not the class – allows us to work with values not referring to any existing object. This is useful, for example, when we have a navigation expression pointing to a class with multiplicity `0..1`. The result of the navigation expression is a value referring to the actual object only if a target object exists. Otherwise, the result is the undefined value.

4.4.1 Operations

There are four different kinds of operations that are specific to object types.

- *Predefined operations*: These are operations which are implicitly defined in OCL for all object types.
- *Attribute operations*: An attribute operation allows access to the attribute value of an object in a given system state.
- *Object operations*: A class may have operations that do not have side effects. These operations are marked in the UML model with the tag *isQuery* [OMG99c, p. 2-25]. In general, OCL expressions could be used to define object operations. The semantics of an object operation is therefore given by the semantics of the associated OCL expression.
- *Navigation operations*: An object may be connected to other objects via association links. A navigation expression allows to follow these links and to retrieve connected objects.

Predefined operations

For all classes $c \in \text{CLASS}$ with object type $t_c = \text{typeOf}(c)$ the operations

$$\text{allInstances}_{t_c} : \rightarrow Set(t_c)$$

are in Ω_C. The semantics is defined as

$$I(\text{allInstances}_{t_c} : \rightarrow Set(t_c)) = \sigma_{\text{CLASS}}(c) \ .$$

This interpretation of allInstances is safe in the sense that its result is always limited to a finite set. The extension of a class is always a finite set of objects. As mentioned in Chapter 2, allInstances could be applied to any type in previous OCL versions, thus allowing potentially unsafe expressions.

Attribute operations

Attribute operations are declared in a model specification by the set ATT_c for each class c. The set contains signatures $a : t_c \rightarrow t$ with a being the name of an attribute defined in the class c. The type of the attribute is t. All attribute operations in ATT_c are elements of Ω_C. The semantics of an attribute operation is a function mapping an object identifier to a value of the attribute domain. An attribute value depends on the current system state.

Definition 4.9 (Semantics of attribute operations)
An attribute signature $a : t_c \rightarrow t$ in Ω_C is interpreted by an attribute value function $I_{\text{ATT}}(a : t_c \rightarrow t) : I(t_c) \rightarrow I(t)$ mapping objects of class c to a value of type t.

$$I_{\text{ATT}}(a : t_c \rightarrow t)(\underline{c}) = \begin{cases} \sigma_{\text{ATT}}(a)(\underline{c}) & \text{if } \underline{c} \in \sigma_{\text{CLASS}}(c), \\ \bot & \text{otherwise.} \end{cases}$$

□

Note that attribute functions are defined for all possible objects. The attempt to access an attribute of a non-existent object results in an undefined value.

Example. Some possible attribute values for employee objects are given below. These attributes are inherited from the parent class *Person*. The example refers to the state shown in Figure 3.3 on page 48. Interpretations

are given for the attributes age : *Person* → *Integer* and email : *Person* → *Set*(*String*).

$$
\begin{aligned}
I(\text{age})(\underline{e}_1) &= 47 \\
I(\text{age})(\underline{e}_2) &= 23 \\
I(\text{age})(\underline{e}_3) &= \bot \\
I(\text{age})(\bot) &= \bot \\
I(\text{email})(\underline{e}_1) &= \emptyset \\
I(\text{email})(\underline{e}_2) &= \{\text{'clark@home.org'}, \text{'clark@work.com'}\} \\
I(\text{email})(\underline{e}_3) &= \bot
\end{aligned}
$$

□

Object operations

Object operations are declared in a model specification. For side effect-free operations the computation can often be described with an OCL expression. The semantics of a side effect-free object operation can then be given by the semantics of the OCL expression associated with the operation. We give a semantics for object operations in Chapter 5 when OCL expressions are introduced.

Example. We have defined an operation `rentalsForDay` in class *Branch* that retrieves all rentals for a given day. The computation required for this operation is given as an OCL expression that selects rentals by comparing their `fromDay` and `untilDay` attributes with the specified day.

```
rentalsForDay(day : String) : Set(Rental) =
  rental->select(r : Rental |
    r.fromDay <= day and day <= r.untilDay)
```

□

Navigation operations

A fundamental concept of OCL is navigation along associations. Navigation operations start from an object of a source class and retrieve all connected objects of a target class. In general, every n-ary association induces a total of $n \cdot (n-1)$ directed navigation operations, because OCL navigation operations only consider two classes of an association at a time. For defining the set of navigation operations of a given class, we have to consider all associations the class is participating in. A corresponding function named participating was defined on page 39.

Definition 4.10 (Syntax of navigation operations)
Let $\mathcal{M}$ be a model

$$\mathcal{M} = (\textsc{Class}, \textsc{Att}_c, \textsc{Op}_c, \textsc{Assoc}, \text{associates}, \text{roles}, \text{multiplicities}, \prec) \ .$$

The set $\Omega_{\text{nav}}(c)$ of navigation operations for a class $c \in \textsc{Class}$ is defined such that for each association $as \in \text{participating}(c)$ with $\text{associates}(as) = \langle c_1, \ldots, c_n \rangle$, $\text{roles}(as) = \langle r_1, \ldots, r_n \rangle$, and $\text{multiplicities}(as) = \langle M_1, \ldots, M_n \rangle$ the following signatures are in $\Omega_{\text{nav}}(c)$.

For all $i, j \in \{1, \ldots, n\}$ with $i \neq j$, $c_i = c$, $t_{c_i} = \text{typeOf}(c_i)$, and $t_{c_j} = \text{typeOf}(c_j)$

i. if $n = 2$ and $M_j - \{0, 1\} = \emptyset$ then $r_{j_{(as,r_i)}} : t_{c_i} \to t_{c_j} \in \Omega_{\text{nav}}(c)$,

ii. if $n > 2$ or $M_j - \{0, 1\} \neq \emptyset$ then $r_{j_{(as,r_i)}} : t_{c_i} \to \mathit{Set}(t_{c_j}) \in \Omega_{\text{nav}}(c)$.

All navigation operations are elements of Ω_C. □

As discussed in Chapter 3, we use unique role names instead of class names for navigation operations in order to avoid ambiguities. The index of the navigation operation name specifies the association to be navigated along as well as the source role name of the navigation path. The result type of a navigation over binary associations is the type of the target class if the multiplicity of the target is given as `0..1` or `1` (i). All other multiplicities allow an object of the source class to be linked with multiple objects of the target class. Therefore, we need a set type to represent the navigation result (ii). Non-binary associations always induce set-valued results since a multiplicity at the target end is interpreted in terms of *all* source objects. However, for a navigation operation, only a single source object is considered. It is not possible in UML to specify a restriction for a single source object using only the concept of multiplicities. The following example shows how this can easily be expressed with an additional OCL constraint.

Example. Below we give the navigation operations of classes *Car* and *CarGroup* (see Figure 2.2 on page 11). Note that navigating the ternary association Maintenance from *Car* to *ServiceDepot* results in a set although the multiplicity of the target end is specified as `0..1`. As mentioned above, this multiplicity specification only applies to pairs of *Car* and *Check* objects in UML.

$$\begin{aligned}\Omega_{\text{nav}}(\mathit{Car}) = \{\ & \text{carGroup}_{(\text{Classification,car})} : \mathit{Car} \to \mathit{CarGroup}, \\ & \text{check}_{(\text{Maintenance,car})} : \mathit{Car} \to \mathit{Set}(\mathit{Check}), \\ & \text{serviceDepot}_{(\text{Maintenance,car})} : \mathit{Car} \to \mathit{Set}(\mathit{ServiceDepot}), \\ & \text{rental}_{(\text{Assignment,car})} : \mathit{Car} \to \mathit{Rental}, \\ & \text{branch}_{(\text{Fleet,car})} : \mathit{Car} \to \mathit{Branch}\ \}\end{aligned}$$

$$\Omega_{\text{nav}}(\mathit{CarGroup}) = \{\ \text{lower}_{(\text{Quality,higher})} : \mathit{CarGroup} \to \mathit{CarGroup},\ \text{higher}_{(\text{Quality,lower})} : \mathit{CarGroup} \to \mathit{CarGroup}\ \}$$

The following OCL invariant uses a navigation operation from *Car* objects to *ServiceDepot* objects in order to specify a restriction on the size of the resulting set.

```
-- A maintenance is done in only one service depot
-- (this cannot be expressed with multiplicities
-- on ternary associations)
context Car inv:
  self.serviceDepot->size <= 1
```

□

Navigation operations are interpreted by navigation functions. Such a function has the effect of first selecting all those links of an association where the source object occurs in the link component corresponding to the role of the source class. The resulting links are then projected onto those objects that correspond to the role of the target class.

Definition 4.11 (Semantics of navigation operations)
The set of objects of class c_j linked to an object $\underline{c}_i$ via association as is defined as

$$L(as)(\underline{c}_i) = \{\underline{c}_j \mid\ (\underline{c}_1, \ldots, \underline{c}_i, \ldots, \underline{c}_j, \ldots, \underline{c}_n) \in \sigma_{\text{Assoc}}(as)\}$$

The semantics of operations in $\Omega_{\text{nav}}(c)$ is then defined as

i. $I(r_{j_{(as,r_i)}} : t_{c_i} \to t_{c_j})(\underline{c}_i) = \begin{cases} \underline{c}_j & \text{if } \underline{c}_j \in L(as)(\underline{c}_i), \\ \bot & \text{otherwise.} \end{cases}$

ii. $I(r_{j_{(as,r_i)}} : t_{c_i} \to \mathit{Set}(t_{c_j}))(\underline{c}_i) = L(as)(\underline{c}_i)$.

□

Example. The following functions show possible navigations between objects and the results. The interpretation is based on the system state presented in Figure 3.3 on page 48.

$$I(\text{manager}_{(\text{Management,managedBranch})} : \mathit{Branch} \to \mathit{Employee})(\underline{b}_1) = \underline{e}_1$$
$$I(\text{managedBranch}_{(\text{Management,manager})} : \mathit{Employee} \to \mathit{Branch})(\underline{e}_1) = \underline{b}_1$$
$$I(\text{managedBranch}_{(\text{Management,manager})} : \mathit{Employee} \to \mathit{Branch})(\underline{e}_2) = \bot$$
$$I(\text{employee}_{(\text{Employment,employer})} : \mathit{Branch} \to \mathit{Set}(\mathit{Employee}))(\underline{b}_1) = \{\underline{e}_1, \underline{e}_2\},$$
$$I(\text{employer}_{(\text{Employment,employee})} : \mathit{Employee} \to \mathit{Branch})(\underline{e}_1) = \underline{b}_1$$

□

4.5 Collection Types

We call a type that allows the aggregation of several values into a single value a complex type. OCL provides the complex types $Set(t)$, $Sequence(t)$, and $Bag(t)$ for describing collections of values of type t. There is also an abstract supertype $Collection(t)$ which describes common properties of these types. The OCL collection types are homogeneous in the sense that all elements of a collection must be of the same type t. This restriction is slightly relaxed by the substitution rule for subtypes in OCL (see Section 4.7). The rule says that the actual elements of a collection must have a type which is a subtype of the declared element type. For example, a $Set(Person)$ may contain elements of type *Customer* or *Employee*.

4.5.1 Syntax and Semantics

Since collection types are parameterized types, we define their syntax recursively by means of type expressions.

Definition 4.12 (Type expressions)
Let $\hat{T}$ be a set of types. The set of type expressions $T_{\text{Expr}}(\hat{T})$ over $\hat{T}$ is defined as follows.

i. If $t \in \hat{T}$ then $t \in T_{\text{Expr}}(\hat{T})$.

ii. If $t \in T_{\text{Expr}}(\hat{T})$ then $Set(t), Sequence(t), Bag(t) \in T_{\text{Expr}}(\hat{T})$.

iii. If $t \in T_{\text{Expr}}(\hat{T})$ then $Collection(t) \in T_{\text{Expr}}(\hat{T})$.

□

The definition says that every type $t \in \hat{T}$ can be used as an element type for constructing a set, sequence, bag, or collection type. Furthermore, these complex types may again be used as element types for constructing other complex types. The recursive definition allows unlimited nesting of type expressions.

In general, the definition of type expressions could be defined to include more kinds of complex types. For example, the object specification language TROLL *light* also defines maps and unions as part of a calculus of complex values [Her95]. A map type can always be represented as a set of pairs (tuples). The concept of a union is, in a limited way, already available as a consequence of subtype polymorphism in OCL. For example, the type $Set(Person)$ can also carry elements of type *Customer* or *Employee*.

For the definition of the semantics of type expressions we make the following conventions. Let $\mathcal{F}(S)$ denote the set of all finite subsets of a given set S, S^* is the set of all finite sequences over S, and $\mathcal{B}(S)$ is the set of all finite multisets (bags) over S.

Definition 4.13 (Semantics of type expressions)
Let $\hat{T}$ be a set of types where the domain of each $t \in \hat{T}$ is $I(t)$. The semantics of type expressions $T_{\text{Expr}}(\hat{T})$ over $\hat{T}$ is defined for all $t \in \hat{T}$ as follows.

i. $I(t)$ is defined as given.

ii. $I(\mathit{Set}(t)) = \mathcal{F}(I(t)) \cup \{\bot\}$,
$I(\mathit{Sequence}(t)) = (I(t))^* \cup \{\bot\}$,
$I(\mathit{Bag}(t)) = \mathcal{B}(I(t)) \cup \{\bot\}$.

iii. $I(\mathit{Collection}(t)) = I(\mathit{Set}(t)) \cup I(\mathit{Sequence}(t)) \cup I(\mathit{Bag}(t))$.

□

In this definition, we observe that the interpretation of the type $\mathit{Collection}(t)$ subsumes the semantics of the set, sequence and bag type. In OCL, the collection type is described as an "abstract" supertype of $\mathit{Set}(t)$, $\mathit{Sequence}(t)$ and $\mathit{Bag}(t)$. This construction greatly simplifies the definition of operations having a similar semantics for each of the concrete collection types. Instead of explicitly repeating these operations for each collection type, they are defined once for $\mathit{Collection}(t)$. Examples for operations which are "inherited" in this way are the size and includes operations which determine the number of elements in a collection or test for the presence of an element in a collection, respectively.

4.5.2 Operations

Constructors

The most obvious way to create a collection value is by explicitly enumerating its element values. We therefore define a set of generic operations which allow us to construct sets, sequences, and bags from an enumeration of element values. For example, the set $\{1, 2, 5\}$ can be described in OCL by the expression `Set{1,2,5}`, the list $\langle 1, 2, 5\rangle$ by `Sequence{1,2,5}`, and the bag $\{\!\{2, 2, 7\}\!\}$ by `Bag{2,2,7}`. A shorthand notation for collections containing integer intervals can be used by specifying lower and upper bounds of the interval. For example, the expression `Sequence{3..6}` denotes the sequence $\langle 3, 4, 5, 6\rangle$. This is only syntactic sugar because the same collection can be described by explicitly enumerating all values of the interval.

Operations for constructing collection values by enumerating their element values are called *constructors*. For types $t \in T_{\text{Expr}}(\hat{T})$ constructors in $\Omega_{T_{\text{Expr}}(\hat{T})}$ are defined below. A parameter list $t \times \cdots \times t$ denotes n ($n \geq 0$) parameters of the same type t. We define constructors mkSet_t, mkSequence_t, and mkBag_t not only for any type t but also for any finite number n of parameters.

- $\text{mkSet}_t : t \times \cdots \times t \to \mathit{Set}(t)$
- $\text{mkSequence}_t : t \times \cdots \times t \to \mathit{Sequence}(t)$
- $\text{mkBag}_t : t \times \cdots \times t \to \mathit{Bag}(t)$

The semantics of constructors is defined for values $v_1, \ldots, v_n \in I(t)$ by the following functions.

- $I(\text{mkSet}_t)(v_1, \ldots, v_n) = \{v_1, \ldots, v_n\}$
- $I(\text{mkSequence}_t)(v_1, \ldots, v_n) = \langle v_1, \ldots, v_n \rangle$
- $I(\text{mkBag}_t)(v_1, \ldots, v_n) = \{\!\{v_1, \ldots, v_n\}\!\}$

Note that constructors having element values as arguments are deliberately defined not to be strict. A collection value therefore may contain undefined values while still being well-defined.

Collection operations

The definition of operations of collection types comprises the set of all operations defined in [OMG99b, pp. 7-35]. Operations common to the types $\mathit{Set}(t)$, $\mathit{Sequence}(t)$, and $\mathit{Bag}(t)$ are defined for the supertype $\mathit{Collection}(t)$. Table 4.4 shows the operation schema for these operations. For all $t \in T_{\text{Expr}}(\hat{T})$, the signatures resulting from instantiating the schema are included in $\Omega_{T_{\text{Expr}}(\hat{T})}$. The right column of the table illustrates the intended set-theoretic interpretation. For this purpose, C, C_1, C_2 are values of type $\mathit{Collection}(t)$, and v is a value of type t.

The operation schema in Table 4.4 can be applied to sets (sequences, bags) by substituting $\mathit{Set}(t)$ ($\mathit{Sequence}(t)$, $\mathit{Bag}(t)$) for all occurrences of type $\mathit{Collection}(t)$. A semantics for the operations in Table 4.4 can be easily defined for each of the concrete collection types $\mathit{Set}(t)$, $\mathit{Sequence}(t)$, and $\mathit{Bag}(t)$. The semantics for the operations of $\mathit{Collection}(t)$ can then be reduced to one of the three cases of the concrete types because every collection type

Signature	Semantics
size : $\mathit{Collection}(t) \to \mathit{Integer}$	$\lvert C\rvert$
count : $\mathit{Collection}(t) \times t \to \mathit{Integer}$	$\lvert C \cap \{v\}\rvert$
includes : $\mathit{Collection}(t) \times t \to \mathit{Boolean}$	$v \in C$
excludes : $\mathit{Collection}(t) \times t \to \mathit{Boolean}$	$v \notin C$
includesAll : $\mathit{Collection}(t) \times \mathit{Collection}(t) \to \mathit{Boolean}$	$C_2 \subseteq C_1$
excludesAll : $\mathit{Collection}(t) \times \mathit{Collection}(t) \to \mathit{Boolean}$	$C_2 \cap C_1 = \emptyset$
isEmpty : $\mathit{Collection}(t) \to \mathit{Boolean}$	$C = \emptyset$
notEmpty : $\mathit{Collection}(t) \to \mathit{Boolean}$	$C \neq \emptyset$
sum : $\mathit{Collection}(t) \to t$	$\sum_{i=1}^{\lvert C\rvert} c_i$

Table 4.4: Operations for type $\mathit{Collection}(t)$

is either a set, a sequence, or a bag. Consider, for example, the operation count : $\mathit{Set}(t) \times t \to \mathit{Integer}$ that counts the number of occurrences of an element v in a set s. The semantics of count is

$$I(\text{count} : \mathit{Set}(t) \times t \to \mathit{Integer})(s, v) = \begin{cases} 1 & \text{if } v \in s, \\ 0 & \text{if } v \notin s, \\ \bot & \text{if } s = \bot. \end{cases}$$

Note that count is not strict. A set may contain the undefined value so that the result of count is 1 if the undefined value is passed as the second argument, for example, $\text{count}(\{\bot\}, \bot) = 1$ and $\text{count}(\{1\}, \bot) = 0$.

For bags (and very similar for sequences), the meaning of count is

$$I(\text{count} : \mathit{Bag}(t) \times t \to \mathit{Integer})(\{\!\{v_1, \ldots, v_n\}\!\}, v) = \begin{cases} 0 & \text{if } n = 0, \\ I(\text{count})(\{\!\{v_2, \ldots, v_n\}\!\}, v) & \text{if } n > 0 \text{ and } v_1 \neq v, \\ I(\text{count})(\{\!\{v_2, \ldots, v_n\}\!\}, v) + 1 & \text{if } n > 0 \text{ and } v_1 = v. \end{cases}$$

As explained before, the semantics of count for values of type $\mathit{Collection}(t)$ can now be defined in terms of the semantics of count for sets, sequences, and bags.

$$I(\text{count} : \mathit{Collection}(t) \times t \to \mathit{Integer})(c, v) = \begin{cases} I(\text{count} : \mathit{Set}(t) \times t \to \mathit{Integer})(c, v) & \text{if } c \in I(\mathit{Set}(t)), \\ I(\text{count} : \mathit{Sequence}(t) \times t \to \mathit{Integer})(c, v) & \text{if } c \in I(\mathit{Sequence}(t)), \\ I(\text{count} : \mathit{Bag}(t) \times t \to \mathit{Integer})(c, v) & \text{if } c \in I(\mathit{Bag}(t)), \\ \bot & \text{otherwise.} \end{cases}$$

Set operations

Operations on sets include the operations listed in Table 4.4. These are inherited from $Collection(t)$. Operations which are specific to sets are shown in Table 4.5 where S, S_1, S_2 are values of type $Set(t)$, B is a value of type $Bag(t)$ and v is a value of type t.

Signature	Semantics
union : $Set(t) \times Set(t) \to Set(t)$	$S_1 \cup S_2$
union : $Set(t) \times Bag(t) \to Bag(t)$	$S \cup B$
intersection : $Set(t) \times Set(t) \to Set(t)$	$S_1 \cap S_2$
intersection : $Set(t) \times Bag(t) \to Set(t)$	$S \cap B$
$-$: $Set(t) \times Set(t) \to Set(t)$	$S_1 - S_2$
symmetricDifference : $Set(t) \times Set(t) \to Set(t)$	$(S_1 \cup S_2) - (S_1 \cap S_2)$
including : $Set(t) \times t \to Set(t)$	$S \cup \{v\}$
excluding : $Set(t) \times t \to Set(t)$	$S - \{v\}$
asSequence : $Set(t) \to Sequence(t)$	
asBag : $Set(t) \to Bag(t)$	

Table 4.5: Operations for type $Set(t)$

Note that the semantics of the operation asSequence is nondeterministic. Any sequence containing only the elements of the source set (in arbitrary order) satisfies the operation specification in OCL.

Bag operations

Operations for bags are shown in Table 4.6. The operation asSequence is nondeterministic also for bags.

Signature	Semantics
union : $Bag(t) \times Bag(t) \to Bag(t)$	$B_1 \cup B_2$
union : $Bag(t) \times Set(t) \to Bag(t)$	$B \cup S$
intersection : $Bag(t) \times Bag(t) \to Bag(t)$	$B_1 \cap B_2$
intersection : $Bag(t) \times Set(t) \to Set(t)$	$B \cap S$
including : $Bag(t) \times t \to Bag(t)$	$B \cup \{\!\{v\}\!\}$
excluding : $Bag(t) \times t \to Bag(t)$	$B - \{\!\{v\}\!\}$
asSequence : $Bag(t) \to Sequence(t)$	
asSet : $Bag(t) \to Set(t)$	

Table 4.6: Operations for type $Bag(t)$

Sequence operations

Sequence operations are displayed in Table 4.7. The intended semantics again is shown in the right column of the table. S, S_1, S_2 are sequences occurring as argument values, v is a value of type t, and i, j are arguments of type *Integer*. The length of sequence S is n. The operator $\circ$ denotes the concatenation of lists, $\pi_i(S)$ projects the ith element of a sequence S, and $\pi_{i,j}(S)$ results in a subsequence of S starting with the ith element up to and including the jth element. The result is $\perp$ if an index is out of range. $S - \langle v \rangle$ produces a sequence equal to S but with all elements equal to v removed. Note that the operations append and including are also defined identically in the OCL standard.

Signature	Semantics
union : $Sequence(t) \times Sequence(t) \rightarrow Sequence(t)$	$S_1 \circ S_2$
append : $Sequence(t) \times t \rightarrow Sequence(t)$	$S \circ \langle e \rangle$
prepend : $Sequence(t) \times t \rightarrow Sequence(t)$	$\langle e \rangle \circ S$
subSequence : $Sequence(t) \times Integer \times Integer \rightarrow Sequence(t)$	$\pi_{i,j}(S)$
at : $Sequence(t) \times Integer \rightarrow t$	$\pi_i(S)$
first : $Sequence(t) \rightarrow t$	$\pi_1(S)$
last : $Sequence(t) \rightarrow t$	$\pi_n(S)$
including : $Sequence(t) \times t \rightarrow Sequence(t)$	$S \circ \langle e \rangle$
excluding : $Sequence(t) \times t \rightarrow Sequence(t)$	$S - \langle e \rangle$
asSet : $Sequence(t) \rightarrow Set(t)$	
asBag : $Sequence(t) \rightarrow Bag(t)$	

Table 4.7: Operations for type $Sequence(t)$

Flattening of collections

Type expressions as introduced in Definition 4.12 allow arbitrarily deep nested collection types. However, nested collections are not really supported in OCL. In fact, "all Collections of Collections are flattened automatically" [OMG99b, p. 7-20]. Unfortunately, the details of this flattening process remain unclear. Catalysis uses flat sets as default when associations with multiplicity * are navigated [DW98]. Nested sets are possible by specifying the type explicitly.

The idea of "flat" sets results from navigating multiple associations where every single navigation results in a set of objects. It is convenient if the result of the whole navigation process can be collected in a single flat set. Figure 4.3 shows an example. In order to retrieve all employees of a department, we first have to navigate the Controls association for getting all projects controlled

by the department. The employees working on each of these projects are determined by traversing the WorksOn association for each project.

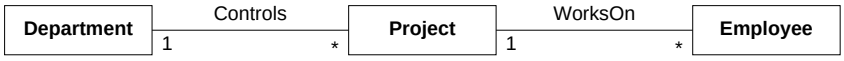


Figure 4.3: Example for navigation along two associations.

All employees of a department `d` can be selected by the OCL expression

```
d.project->collect(p : Project | p.employee)
```

Because this kind of expression occurs so often, OCL also allows the shorthand notation `d.project.employee` for this expression. The result type of this expression is $Bag(Set(Employee))$ which is "automatically" flattened into $Bag(Employee)$.

We pursue the following approach for giving a precise meaning to collection flattening. First, we keep nested collection types because they do not only make the type system more orthogonal, but they are also necessary for describing the input of the flattening process. Second, we define flattening by means of an explicit function making the effect of the flattening process clear. We can interpret OCL expressions like the one given above as a shorthand notation in concrete syntax which would expand in abstract syntax to an expression with an explicit flattening function.

Flattening in OCL does apply to all collection types. We have to consider all possible combinations first. Table 4.8 shows all possibilities for combining *Set*, *Bag*, and *Sequence* into a nested collection type. For each of the different cases, the collection type resulting from flattening is shown in the right column. Note that the element type t can be any type. In particular, if t is also a collection type the indicated rules for flattening can be applied recursively until the element type of the result is a non-collection type.

A signature schema for a flatten operation that removes one level of nesting can be defined as

$$\text{flatten} : C_1(C_2(t)) \rightarrow C_1(t)$$

where C_1 and C_2 denote any collection type name *Set*, *Sequence*, or *Bag*. The meaning of the flatten operations can be defined by the following generic iterate expression. The semantics of OCL iterate expressions is defined in the next chapter in Section 5.1.2.

```
<collection-of-type-C1(C2(t))>->iterate(e1 : C2(t);
  acc1 : C1(t) = C1{} |
  e1->iterate(v : t;
    acc2 : C1(t) = acc1 |
    acc2->including(v)))
```

Nested collection type	Type after flattening
$Set(Sequence(t))$	$Set(t)$
$Set(Set(t))$	$Set(t)$
$Set(Bag(t))$	$Set(t)$
$Bag(Sequence(t))$	$Bag(t)$
$Bag(Set(t))$	$Bag(t)$
$Bag(Bag(t))$	$Bag(t)$
$Sequence(Sequence(t))$	$Sequence(t)$
$Sequence(Set(t))$	$Sequence(t)$
$Sequence(Bag(t))$	$Sequence(t)$

Table 4.8: Flattening of nested collections.

The following example shows how this expression schema is instantiated for a bag of sets of integers, that is, $C_1 = Bag$, $C_2 = Set$, and $t = Integer$. The result of flattening the value `Bag{Set{3,2},Set{1,2,4}}` is `Bag{1,2,2,3,4}`.

```
Bag{Set{3,2},Set{1,2,4}}->iterate(e1 : Set(Integer);
  acc1 : Bag(Integer) = Bag{} |
  e1->iterate(v : Integer;
    acc2 : Bag(Integer) = acc1 |
    acc2->including(v)))
```

It is important to note that flattening sequences of sets and bags (see the last two rows in Table 4.8) is potentially nondeterministic. For these two cases, the flatten operation would have to map each element of the (multi-) set to a distinct position in the resulting sequence, thus imposing an order on the elements which did not exist in the first place. Since there are types (e.g. object types) which do not define an order on their domain elements, there is no obvious mapping for these types. Fortunately, these problematic cases do not occur in standard navigation expressions. Furthermore, these kinds of collections can be flattened if the criteria for ordering the elements is explicitly specified.

4.6 Special Types

Special types in OCL that do not fit into the categories discussed so far are *OclAny*, *OclType*, *OclExpression*, and *OclState*.

- *OclAny* is the supertype of all other types except for the collection types. The exception has been introduced in UML 1.3 because it considerably simplifies the type system [CKM+99a]. A simple set inclusion semantics for subtype relationships as proposed in the next section

would not be possible due to cyclic domain definitions if *OclAny* were the supertype of *Set*(*OclAny*).

- *OclType* adds a further level of abstraction to the type system. This type introduces a meta-level which is placed above the level of ordinary types. Values of *OclType* are all OCL types. The probably most important operation on *OclType* is allInstances. When we apply this operation to an object type, the result is the set of all objects of this type currently existing in a given system state. There are a few more operations defined in OCL dealing with *OclType* values. For example, the attributes, association ends, and operation names of a type can be retrieved. However, these operations are of little benefit since they only return part of the information as sets of strings and thus only provide very limited access to the metamodel [BH00].

- *OclExpression* is the type of OCL expressions. In the standard OCL document, it is mainly used to specify signatures of operations which are based on the iterate construct [OMG99b]. Chapter 5 shows that the syntax of expressions can easily be defined without the need for an extra expression type.

- *OclState* is a type very similar to an enumeration type. It is only used in the operation oclInState for referring to state names in a state machine. There are no operations defined on this type.

We conclude that *OclAny* is an important part of the OCL type system whereas *OclType* adds more complexity than benefit. The important operation allInstances has already been defined as an ordinary operation schema on object types in Section 4.4. Therefore, we do not consider *OclType* in the following. The types *OclExpression* and *OclState* can be dealt with in other ways.

Definition 4.14 (Type *OclAny*)
The set of special types is $T_S = \{OclAny\}$.

Let $\hat{T}$ be the set of basic, enumeration, and object types $\hat{T} = T_B \cup T_E \cup T_C$. The domain of OclAny is given as $I(OclAny) = \left(\bigcup_{t \in \hat{T}} I(t)\right) \cup \{\bot\}$. □

Operations on *OclAny* include equality (=) and inequality (<>) which already have been defined for all types in Section 4.2.4. The operations oclIsKindOf, oclIsTypeOf, and oclAsType expect a type as argument. We define them as part of the OCL expression syntax in the next chapter. The operation oclIsNew is only allowed in postconditions and will therefore be discussed in the next chapter.

4.7 Type Hierarchy

The type system of OCL supports inclusion polymorphism [CW85] by introducing the concept of a *type hierarchy*. The type hierarchy is used to define the notion of *type conformance*. Type conformance is a relationship between two types. A valid OCL expression is an expression in which all the types conform [OMG99b, p. 7-9]. The consequence of type conformance can be loosely stated as: a value of a conforming type B may be used wherever a value of type A is required.

The type hierarchy reflects the subtype/supertype relationship between types. The following relationships are defined in OCL.

1. *Integer* is a subtype of *Real*.
2. All types, except for the collection types, are subtypes of *OclAny*.
3. $Set(t)$, $Sequence(t)$, and $Bag(t)$ are subtypes of $Collection(t)$.
4. The hierarchy of types introduced by UML model elements mirrors the generalization hierarchy in the UML model.

The last rule is not explicitly stated in [OMG99b], but seems to reflect the intended meaning. Type conformance is a relation which is identical to the subtype relation introduced by the type hierarchy. The relation is reflexive and transitive.

Definition 4.15 (Type hierarchy)
Let T be a set of types and T_C a set of object types with $T_C \subset T$. The relation $\leq$ is a partial order on T and is called the *type hierarchy* over T. The type hierarchy is defined for all $t, t', t'' \in T$ and all $t_c, t'_c \in T_C$ as follows.

i. $\leq$ is (a) reflexive, (b) transitive, and (c) antisymmetric:
   (a) $t \leq t$
   (b) $t'' \leq t' \wedge t' \leq t \implies t'' \leq t$
   (c) $t' \leq t \wedge t \leq t' \implies t = t'$.

ii. $Integer \leq Real$.

iii. $t \leq OclAny$ for all $t \in (T_B \cup T_E \cup T_C)$.

iv. $Set(t) \leq Collection(t)$,
$Sequence(t) \leq Collection(t)$, and
$Bag(t) \leq Collection(t)$.

v. If $t' \leq t$ then $Set(t') \leq Set(t)$, $Sequence(t') \leq Sequence(t)$, $Bag(t') \leq Bag(t)$, and $Collection(t') \leq Collection(t)$.

vi. If $\text{classOf}(t'_c) \prec \text{classOf}(t_c)$ then $t'_c \leq t_c$.

□

If a type t' is a subtype of another type t (i.e. $t' \leq t$), we say that t' *conforms* to t. Type conformance is associated with the principle of substitutability. A value of type t' may be used wherever a value of type t is expected. This rule is defined more formally in Section 5.1 which defines the syntax and semantics of expressions.

The principle of substitutability and the interpretation of types as sets suggest that the type hierarchy should be defined as a subset relation on the type domains. Hence, for a type t' being a subtype of t, we postulate that the interpretation of t' is a subset of the interpretation of t. It follows that every operation ω accepting values of type t has the same semantics for values of type t', since $I(\omega)$ is already well-defined for values in $I(t')$.

Proposition 4.1 (Semantics of a type hierarchy)
If $t' \leq t$ then $I(t') \subseteq I(t)$ for all types $t', t \in T$.

The proof goes by induction on the structure of the relation $\leq$:

i. Inclusion in set systems is a partial order.

ii. $I(Integer) \subseteq I(Real)$ because $(\mathbb{Z} \cup \{\bot\}) \subseteq (\mathbb{R} \cup \{\bot\})$.

iii. Follows from Definition 4.14.

iv. $I(Set(t)) \subseteq I(Collection(t))$, $I(Sequence(t)) \subseteq I(Collection(t))$ and $I(Bag(t)) \subseteq I(Collection(t))$ are a consequence of Definition 4.13.

v. $I(Set(t')) \subseteq I(Set(t))$, $I(Sequence(t')) \subseteq I(Sequence(t))$, $I(Bag(t')) \subseteq I(Bag(t))$ and $I(Collection(t')) \subseteq I(Collection(t))$ follow from Definition 4.13.

vi. $I(c') \subseteq I(c)$ follows directly from the definitions of object identifiers (Definition 3.10 on page 44) and object types (Definition 4.8 on page 62).

□

4.8 Data Signature

We now have available all elements necessary to define the final data signature for OCL expressions. The signature provides the basic set of syntactic elements for building expressions. It defines the syntax and semantics of types, the type hierarchy, and the set of operations defined on types.

Definition 4.16 (Data signature)
Let $\hat{T}$ be the set of non-collection types: $\hat{T} = T_B \cup T_E \cup T_C \cup T_S$. The syntax of a data signature over an object model $\mathcal{M}$ is a structure $\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$ where

i. $T_{\mathcal{M}} = T_{\text{Expr}}(\hat{T})$,

ii. $\leq$ is a type hierarchy over $T_{\mathcal{M}}$,

iii. $\Omega_{\mathcal{M}} = \Omega_{T_{\text{Expr}}(\hat{T})} \cup \Omega_B \cup \Omega_E \cup \Omega_C \cup \Omega_S$.

The semantics of $\Sigma_{\mathcal{M}}$ is a structure $I(\Sigma_{\mathcal{M}}) = (I(T_{\mathcal{M}}), I(\leq), I(\Omega_{\mathcal{M}}))$ where

i. $I(T_{\mathcal{M}})$ assigns each $t \in T_{\mathcal{M}}$ an interpretation $I(t)$.

ii. $I(\leq)$ implies for all types $t', t \in T_{\mathcal{M}}$ that $I(t') \subseteq I(t)$ if $t' \leq t$.

iii. $I(\Omega_{\mathcal{M}})$ assigns each operation $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_{\mathcal{M}}$ a total function $I(\omega) : I(t_1) \times \cdots \times I(t_n) \to I(t)$.

□

4.9 Extensions

In this section, we discuss possible extensions of the OCL type system. One might argue that we already introduced some extensions previously, for example, by adding operations allowing to test for undefinedness, and a flattening operation for collections. However, we consider these extensions necessary to fill gaps resulting from under-specification in the OCL documentation. There may be other approaches, but for a sound OCL definition *some* approach has to be chosen.

The extensions presented in this section are optional and orthogonal in the sense that they can easily be integrated with the existing type system. The first extension introducing tuple types for building aggregate structures is presented in Section 4.9.1. Association types are proposed in Section 4.9.2. In Section 4.9.3, we discuss user-defined data types.

4.9.1 Tuple Types

There is no complex type in OCL which would allow the ad hoc aggregation of values of unrelated types. Although classes and therefore object types provide aggregation by means of attributes, the number of classes is fixed in the model. New classes cannot be created with OCL. A record or tuple type is a fundamental concept in most semantic and object-oriented data models (e.g., extended Entity-Relationship models) and logical data models (e.g., the relational data model). It is required for expressing structured and complex query results. For example, the query "*Get the set of all branches together with their employees*" has the result type *Set*(*Tuple*(*Branch, Set*(*Employee*))). Each value of the result set is a tuple containing a branch object and a set of associated employee objects. This query cannot be expressed with standard OCL. We therefore propose to add a tuple type $\mathit{Tuple}(t_1, \ldots, t_n)$ to lift this restriction by extending the definition of type expressions. This extension is straightforward and has no negative impact on the standard types.

Definition 4.17 (Type expressions (extended))
Let $\hat{T}$ be a set of types. The set of type expressions $T_{\text{Expr}}(\hat{T})$ over $\hat{T}$ defined in Definition 4.12 is extended as follows.

iv. If $t_1, \ldots, t_n \in T_{\text{Expr}}(\hat{T})$ then $\mathit{Tuple}(t_1, \ldots, t_n) \in T_{\text{Expr}}(\hat{T})$.

The semantics of a tuple type expression is

$$I(\mathit{Tuple}(t_1, \ldots, t_n)) = I(t_1) \times \cdots \times I(t_n) \cup \{\bot\}.$$

□

Now every type $t \in \hat{T}$ can also be used as an element type of a tuple. A constructor for a tuple is an operation

$$\text{mkTuple} : t_1 \times \cdots \times t_n \to \mathit{Tuple}(t_1, \ldots, t_n) \ .$$

The semantics of tuple constructors is defined for values $v_i \in I(t_i)$ with $i = 1, \ldots, n$ by the following function.

$$I(\text{mkTuple})(v_1, \ldots, v_n) = (v_1, \ldots, v_n)$$

Since OCL does not provide a tuple type, we propose the following notation for tuple constructors. The concrete syntax is similar to the OCL syntax for collection types. Additionally, components of a tuple type may optionally be named by a label a_i for each ith component. For example, a tuple with

three components of type *Integer*, *String*, and *Boolean* can be constructed with the expression

```
Tuple{3, 'apple', true}
```

or, using component labels, with an expression like

```
Tuple{number:3, fruit:'apple', flag:true} .
```

In abstract syntax, both expressions are equally mapped to the operation

$$\text{mkTuple} : \mathit{Integer} \times \mathit{String} \times \mathit{Boolean} \to \mathit{Tuple}(\mathit{Integer}, \mathit{String}, \mathit{Boolean}) \ .$$

Example. Figure 4.4 shows a simple class diagram that we use as an example for illustrating the application of OCL expressions with tuple types.

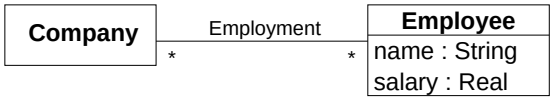


Figure 4.4: Class diagram used for illustrating tuple types

The following standard OCL expression selects for a given company `c` the names of all of its employees. The result type of the expression is *Bag*(*String*).

```
context c : Company:
  c.employee->collect(e | e.name)
```

It is not possible in standard OCL to retrieve the names of employees *together* with their salary in a single structure. However, with tuple types being added, we can express this with the following query. The result type now is *Bag*(*Tuple*(*String, Real*)).

```
context c : Company:
  c.employee->collect(e | Tuple{e.name, e.salary})
```

We can achieve a more concise notation by introducing the following shorthand notation on the level of concrete syntax. The syntax of `collect` is extended to allow a list of expressions. If more than one expression is given, the tuple constructor can be omitted. In this case, the constructor is added implicitly. With this rule, the previous example can be written more briefly like this:

```
context c : Company:
  c.employee->collect(e | e.name, e.salary)
```

□

Tuple operations

An essential operation for tuple types is the projection of a tuple value onto one of its components. For a tuple type with n components, we therefore define n projection operations called at_i ($i = 1, \ldots, n$). Each operation selects the ith component, respectively.

- $\mathrm{at}_i : \mathit{Tuple}(t_1, \ldots, t_n) \to t_i$
- $I(\mathrm{at}_i : \mathit{Tuple}(t_1, \ldots, t_n) \to t_i)(v_1, \ldots, v_n) = v_i$

An element of a tuple with labeled components can be accessed by specifying its label. The name of the label a_i is used as operation symbol.

- $\mathrm{a}_i : \mathit{Tuple}(t_1, \ldots, t_n) \to t_i$
- $I(\mathrm{a}_i : \mathit{Tuple}(t_1, \ldots, t_n) \to t_i)(v_1, \ldots, v_n) = v_i$

For the concrete syntax of tuple operations, we propose a notation that follows the style of OCL. Since selecting a tuple component by its position is similar to the selection of a sequence element, we will use the syntax of the at operation for this purpose. A selection of a tuple component by its label is very similar to accessing an attribute value of an object. We therefore reuse this notation for the tuple component projection by name. Hence, the two expressions

```
Tuple{3, 'apple', true}->at(2)
```

and

```
Tuple{number:3, fruit:'apple', flag:true}.fruit
```

have the same result `'apple'`. Note that the argument of `at` in the above example must be a constant expression. This is required for a correct static type analysis of the whole expression. The integer argument specifies the index i that is used to map the expression to one of the operations $\mathrm{a}_i : \mathit{Tuple}(t_1, \ldots, t_n) \to t_i$. For determining the correct operation, the index must be known before an evaluation of the expression is possible. The problem becomes obvious if we consider an alternative definition of the at operation which uses an extra argument of type *Integer* to specify the component position: $\mathrm{at} : \mathit{Tuple}(t_1, \ldots, t_n) \times \mathit{Integer} \to ?$. In this case, the result type depends on the interpretation of the integer argument. Even worse, if the integer value is undefined or is outside the interval boundaries, the result type is also undefined.

Subtyping

The type hierarchy is extended for tuple types by the usual rules for record types [CW85]. A tuple type A is a subtype of another tuple type B if A has at least all the components of B, and every common component type of A is a subtype of the corresponding component type in B:

$$Tuple(t'_1, \ldots, t'_n, \ldots, t'_m) \leq Tuple(t_1, \ldots, t_n) \text{ iff } t'_i \leq t_i \text{ for } i = 1, \ldots, n \ .$$

4.9.2 Association Types

The central concepts of UML static structure diagrams are classes and the relationships between them. While classes directly map to object types (see Section 4.4), there is no such correspondence between associations and types. Associations do only introduce operations allowing the navigation from one object to associated objects. While this is sufficient for binary associations, the expressiveness of this approach has limitations for associations with degree greater than two. Irrespective of the degree of an association, a navigation expression can only refer to the relationship between at most two classes. Given an object of each associated class, it is thus not possible to determine whether there is a link in which all these objects are participating.

Consider, for example, a ternary association R between classes A, B and C as it is shown in Figure 4.5. From an object of class A, we can navigate to either the set of related objects of class B by using the role name rb, or to related objects of class C with the role name rc. Both operations are independent of each other. If we have two objects of classes B and C which are linked to an A object, there is no way to find out at the same time whether the B and C objects are also linked to each other. For example, if we have three triples $(a', b, c), (a, b', c), (a, b, c')$ representing links in R, we can find out by using navigation expressions that there are indeed combinations of $(a, b), (b, c)$ and (a, c) in R. However, we obviously cannot deduce the existence of a link (a, b, c) from that.

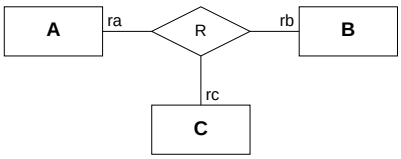


Figure 4.5: Ternary association

Basically, the OCL mechanism of navigation only allows a simplified view of n-ary associations. Figure 4.6 illustrates this view where the association R

is replaced by three binary associations R1, R2, and R3. It is well-known from Entity-Relationship modeling that the two models shown in Figure 4.5 and Figure 4.6 are not equivalent with respect to expressiveness [EN94]. In general, a ternary association represents more information than three binary associations. As shown above, the existence of links (a, b), (b, c), and (a, c) in the second model does not necessarily imply the existence of a link (a, b, c) in the first model.

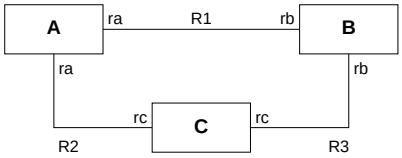


Figure 4.6: View of a ternary association as implied by OCL navigation expressions

OCL navigation operations are a concept for getting information about binary association instances. For links in non-binary associations, that is n-ary associations with $n > 2$, a navigation expression only allows to examine at most two related objects at a time but never the link as a whole. In [GR99], we therefore have proposed – as an extension to OCL – a predicate $R(a, b, c)$ that just provides this functionality. The additional expressiveness brought by this extension is essential for achieving the goal of completeness. By completeness we mean the following property of a constraint language: all structural aspects of a system state are generally accessible to expressions of the constraint language.

In the following, we present a systematic and general approach to satisfy the need for a more powerful OCL concept for associations. The idea is based on the observation that both classifiers and relationships are first-class concepts in UML. Consequently, we introduce types for associations the same way as we have done for classes. The domain of an association type is the set of links that may exist in a system state. There are a number of useful operations for association types. Besides the already mentioned check for link existence, one could easily, for example, write a constraint which guarantees that an association is a subset of another association between the same set of classes.

The syntax of association types and their operations is defined by a signature $\Sigma_A = (T_A, \Omega_A)$. T_A is the set of association types, and Ω_A is the set of signatures describing operations on association types.

Definition 4.18 (Association types)
Let $\mathcal{M}$ be an object model with a set Assoc of association names. The set T_A of association types is defined such that for each association $as \in$

Assoc there is a type $t_a \in T_A$ having the same name as the association. The domain of an association type t_a is $I(t_a) = I_{\text{Assoc}}(as) \cup \{\perp\}$. □

A value of an association type denotes a link of the association. The undefined value $\perp$ denotes a non-existing link. The interpretation of associations as sets of links was introduced in Definition 3.11 on page 46. The interpretation is based on the Cartesian product of the sets of object identifiers of the participating classes.

Operations

We briefly discuss some interesting operations on association types by means of examples without giving formal definitions. All examples refer to the class diagram in Figure 4.5.

An object reference in a link can be accessed by the role name of the object. This can be considered a navigation from a link to a connected object. The following example is a solution to the problem of checking for the existence of a link in a ternary association mentioned at the beginning of this section. The operation `R.allInstances` is similar to the operation allInstances for object types. It results in the set of all links of association R.

```
-- given three objects a, b, and c, check for the
-- existence of a ternary link between them
R.allInstances->exists(r : R |
  r.ra = a and r.rb = b and r.rc = c)
```

Analogously, we can navigate from an object to all links referencing this object. In the following example, the expression `a.R` has type $Set(R)$ and results in the set of links of association R referencing object a.

```
a.R->collect(r : R | Tuple{r.b, r.c})
```

Using the previously introduced tuple type, we can retrieve all pairs of B and C objects connected simultaneously to object a. The result type of the complete expression is $Bag(Tuple(B,C))$. For a more concrete example, substitute classes A, B, and C with *Order*, *Part*, and *Supplier*. Then we could produce for a given order a report containing all parts of the order together with the supplier of each part. In standard OCL with its binary navigation paths, the set of parts and the set of suppliers of an order can only be separately determined. The relationship between parts and suppliers *within the same* order is not accessible.

Links are structurally equivalent to tuples containing object references. The association type R could therefore be identified with the collection type

Set(*Tuple*(*A*,*B*,*C*)). This allows another more concise approach to the first example from above.

```
R.allInstances->includes(Tuple{a,b,c})
```

The following constraint specifies a fundamental property of associations. If two links connect the same objects then these links are identical.

```
context R inv:
  R.allInstances->forAll(
    l1, l2 : R |
      l1.ra = l2.ra and l1.rb = l2.rb and l1.rc = l2.rc
      implies
      l1 = l2)
```

This constraint can be expressed almost identically with tuple types for links.

```
-- same as above using tuple types for links
context R inv:
  R.allInstances->forAll(
    l1, l2 : Tuple(ra : A, rb : B, rc : C) |
      l1.ra = l2.ra and l1.rb = l2.rb and l1.rc = l2.rc
      implies
      l1 = l2)
```

4.9.3 User-defined Data Types

In many situations the predefined types are not sufficient for an adequate model of the problem domain. Some domains require specific data types for modeling addresses, time, geometric coordinates, etc. The common characteristics of these kinds of information is that they can easily be described by a combination of simple types and a set of operations specific to the type at hand. In UML, the class provides a similar concept for describing structured object types. However, an object of a class always has an identity and a state that may change over time. This does not apply for addresses or time values. This distinction between objects and data values is fundamental in modeling [EDS93, EJDS94, GW91, Her95, VHG+93]. Thus different concepts are required for modeling these different aspects.

UML provides a DataType model element in the metamodel. This model element specifies the signature of a data type but does not expose its realization. This corresponds to the definition of an abstract data type. A data type in UML may only contain side effect-free operations but no attributes ([OMG99e, p. 2-30] and [RJB98, p. 247]). In [WK98, p. 82], the

concept of a *utility* class is applied for modeling a *Date* type including attributes for month, day and year. However, this seems to be a misuse of the utility class concept since, according to [RJB98, p. 496]), the attributes of a utility class are just global variables. This concept is only provided for compatibility with non-object-oriented programming languages.

We define the signature of user-defined data types leaving the semantics to an external specification mechanism. The syntax of data types and their operations is defined by a signature $\Sigma_D = (T_D, \Omega_D)$. T_D is the set of user-defined data types. Ω_D is the set of signatures describing operations over user-defined data types.

Figure 4.7 gives an example of a user-defined data type *Date* in UML notation. For the type $Date \in T_D$ we have the following four operation signatures in Ω_D.

$$\begin{aligned} \text{isBefore} &: Date \times Date \to Boolean \\ \text{isAfter} &: Date \times Date \to Boolean \\ \text{day} &: Date \to Integer \\ = &: Date \times Date \to Boolean \end{aligned}$$

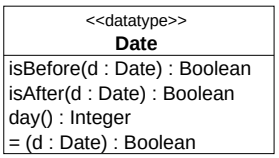


Figure 4.7: A user-defined data type *Date* in UML notation

Summary

In this chapter, we gave a formal definition of the OCL type system. We started by defining basic, enumeration, and object types. Complex types were introduced with collection types. We declared *OclAny* a special type and defined a type hierarchy based on a subtype relation. In the last section we presented some possible extensions to the standard type system. Extensions included tuple types, association types, and user-defined data types.

Based on the standard types we finally defined the data signature $\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$. A data signature describes for a given object model $\mathcal{M}$ the set of types, the type hierarchy, and the set of operations over these types. The data signature directly provides the input for the next chapter defining OCL expressions.

Chapter 5

OCL Expressions and Constraints

The core of OCL is given by an expression language. Expressions can be used in various contexts, for example, to define constraints such as class invariants and pre-/postconditions on operations. In this chapter, we formally define the syntax and semantics of OCL expressions, and give precise meaning to notions like context, invariant, and pre-/postconditions.

Section 5.1 defines the abstract syntax and semantics of OCL expressions and shows how other OCL constructs can be derived from this language core. The context of expressions and other important concepts such as invariants, queries, and shorthand notations are discussed. Section 5.2 defines the meaning of operation specifications with pre- and postconditions. Finally, the expressiveness of OCL is discussed in Section 5.3.

5.1 Expressions

In this section, we define the syntax and semantics of expressions. The definition of expressions is based upon the data signature we developed in the previous chapter. A data signature $\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$ provides a set of types $T_{\mathcal{M}}$, a relation $\leq$ on types reflecting the type hierarchy, and a set of operations $\Omega_{\mathcal{M}}$. The signature contains the initial set of syntactic elements upon which we build the expression syntax.

5.1.1 Syntax of Expressions

We define the syntax of expressions inductively so that more complex expressions are recursively built from simple structures. For each expression the set of free occurrences of variables is also defined.

Definition 5.1 (Syntax of expressions)
Let $\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$ be a data signature over an object model $\mathcal{M}$. Let $\mathrm{Var} = \{\mathrm{Var}_t\}_{t \in T_{\mathcal{M}}}$ be a family of variable sets where each variable set is indexed by a type t. The syntax of expressions over the signature $\Sigma_{\mathcal{M}}$ is given by a set $\mathrm{Expr} = \{\mathrm{Expr}_t\}_{t \in T_{\mathcal{M}}}$ and a function $\mathrm{free} : \mathrm{Expr} \to \mathcal{F}(\mathrm{Var})$ that are defined as follows.

i. If $v \in \mathrm{Var}_t$ then $\boldsymbol{v} \in \mathrm{Expr}_t$ and $\mathrm{free}(v) := \{v\}$.

ii. If $v \in \mathrm{Var}_{t_1}, e_1 \in \mathrm{Expr}_{t_1}, e_2 \in \mathrm{Expr}_{t_2}$ then $\mathbf{let}\ \boldsymbol{v = e_1}\ \mathbf{in}\ \boldsymbol{e_2} \in \mathrm{Expr}_{t_2}$ and $\mathrm{free}(\mathrm{let}\ v = e_1\ \mathrm{in}\ e_2) := \mathrm{free}(e_2) - \{v\}$.

iii. If $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_{\mathcal{M}}$ and $e_i \in \mathrm{Expr}_{t_i}$ for all $i = 1, \ldots, n$ then $\boldsymbol{\omega(e_1, \ldots, e_n)} \in \mathrm{Expr}_t$ and $\mathrm{free}(\omega(e_1, \ldots, e_n)) := \mathrm{free}(e_1) \cup \cdots \cup \mathrm{free}(e_n)$.

iv. If $e_1 \in \mathrm{Expr}_{\mathrm{Boolean}}$ and $e_2, e_3 \in \mathrm{Expr}_t$ then $\mathbf{if}\ \boldsymbol{e_1}\ \mathbf{then}\ \boldsymbol{e_2}\ \mathbf{else}\ \boldsymbol{e_3}\ \mathbf{endif} \in \mathrm{Expr}_t$ and $\mathrm{free}(\mathrm{if}\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3\ \mathrm{endif}) := \mathrm{free}(e_1) \cup \mathrm{free}(e_2) \cup \mathrm{free}(e_3)$.

v. If $e \in \mathrm{Expr}_t$ and $t' \leq t$ or $t \leq t'$ then $\boldsymbol{(e\ \mathbf{asType}\ t')} \in \mathrm{Expr}_{t'}$, $\boldsymbol{(e\ \mathbf{isTypeOf}\ t')} \in \mathrm{Expr}_{\mathrm{Boolean}}$, $\boldsymbol{(e\ \mathbf{isKindOf}\ t')} \in \mathrm{Expr}_{\mathrm{Boolean}}$ and $\mathrm{free}((e\ \mathrm{asType}\ t')) := \mathrm{free}(e)$, $\mathrm{free}((e\ \mathrm{isTypeOf}\ t')) := \mathrm{free}(e)$, $\mathrm{free}((e\ \mathrm{isKindOf}\ t')) := \mathrm{free}(e)$.

vi. If $e_1 \in \mathrm{Expr}_{Collection(t_1)}$, $v_1 \in \mathrm{Var}_{t_1}, v_2 \in \mathrm{Var}_{t_2}$, and $e_2, e_3 \in \mathrm{Expr}_{t_2}$ then $\boldsymbol{e_1 \to \mathbf{iterate}(v_1; v_2 = e_2 \mid e_3)} \in \mathrm{Expr}_{t_2}$ and $\mathrm{free}(e_1 \to \mathrm{iterate}(v_1; v_2 = e_2 \mid e_3)) := (\mathrm{free}(e_1) \cup \mathrm{free}(e_2) \cup \mathrm{free}(e_3)) - \{v_1, v_2\}$.

An expression of type t' is also an expression of a more general type t. For all $t' \leq t$: if $e \in \mathrm{Expr}_{t'}$ then $e \in \mathrm{Expr}_t$. □

A variable expression (i) refers to the value of a variable. Variables (including the special variable `self`) may be introduced by the context of an expression, as part of an iterate expression, and by a let expression. Let expressions (ii) do not add to the expressiveness of OCL but help to avoid repetitions of common sub-expressions. Operation expressions (iii) apply an operation from $\Omega_{\mathcal{M}}$. The set of operations includes:

- predefined data operations: `+`, `-`, `*`, `<`, `>`, `size`, `max`
- attribute operations: `self.age`, `e.salary`
- side effect-free operations defined by a class: `b.rentalsForDay(...)`

- navigation by role names: `self.employee`
- constants: `25`, `'aString'`

As demonstrated by the examples, an operation expression may also be written in OCL path syntax as $e_1.\omega(e_2, \ldots, e_n)$. This notational style is common in many object-oriented languages. It emphasizes the role of the first argument as the "receiver" of a "message". If e_1 denotes a collection value, an arrow symbol is used in OCL instead of the period: $e_1 \rightarrow \omega(e_2, \ldots, e_n)$. Collections may be bags, sets, or lists. An if-expression (iv) provides an alternative selection of two expressions depending on the result of a condition given by a boolean expression.

An asType expression (v) can be used in cases where static type information is insufficient. It corresponds to the `oclAsType` operation in OCL and can be understood as a cast of a source expression to an equivalent expression of a (usually) more specific target type. The target type must be related to the source type, that is, one must be a subtype of the other. The isTypeOf and isKindOf expressions correspond to the `oclIsTypeOf` and `oclIsKindOf` operations, respectively. An expression (e isTypeOf t') can be used to test whether the type of the value resulting from the expression e has the type t' given as argument. An isKindOf expression (e isTypeOf t') is not as strict in that it is sufficient for the expression to become true if t' is a supertype of the type of the value of e. Note that OCL defines these type casts and tests as operations with parameters of type *OclType*. In contrast to OCL, we technically define them as first class expressions which has the benefit that we do not need the metatype *OclType*. Thus the type system is kept simple while preserving compatibility with standard OCL syntax.

An iterate expression (vi) is a general loop construct which evaluates an argument expression e_3 repeatedly for all elements of a collection which is given by a source expression e_1. Each element of the collection is bound in turn to the variable v_1 for each evaluation of the argument expression. The argument expression e_3 may contain the variable v_1 to refer to the current element of the collection. The result variable v_2 is initialized with the expression e_2. After each evaluation of the argument expression e_3, the result is bound to the variable v_2. The final value of v_2 is the result of the whole iterate expression.

The iterate construct is probably the most important kind of expression in OCL. Many other OCL constructs (such as `select`, `reject`, `collect`, `exists`, `forAll`, and `isUnique`) can be equivalently defined in terms of an iterate expression (see Section 5.1.3).

Following the principle of substitutability, the syntax of expressions is defined such that wherever an expression $e \in \text{Expr}_t$ is expected as part of

another expression, an expression with a more special type $t', (t' \leq t)$ may be used. In particular, operation arguments and variable assignments in let and iterate expressions may be given by expressions of more special types.

5.1.2 Semantics of Expressions

The semantics of expressions is made precise in the following definition. A context for evaluation is given by an environment $\tau = (\sigma, \beta)$ consisting of a system state σ and a variable assignment $\beta : \mathrm{Var}_t \to I(t)$. A system state σ provides access to the set of currently existing objects, their attribute values, and association links between objects. A variable assignment β maps variable names to values.

Definition 5.2 (Semantics of expressions)
Let Env be the set of environments $\tau = (\sigma, \beta)$. The semantics of an expression $e \in \mathrm{Expr}_t$ is a function $I[\![\, e \,]\!] : \mathrm{Env} \to I(t)$ that is defined as follows.

i. $I[\![\, v \,]\!](\tau) = \beta(v)$.

ii. $I[\![\, \text{let } v = e_1 \text{ in } e_2 \,]\!](\tau) = I[\![\, e_2 \,]\!](\sigma, \beta\{v/I[\![\, e_1 \,]\!](\tau)\})$.

iii. $I[\![\, \omega(e_1, \ldots, e_n) \,]\!](\tau) = I(\omega)(\tau)(I[\![\, e_1 \,]\!](\tau), \ldots, I[\![\, e_n \,]\!](\tau))$.

iv. $$I[\![\, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ endif} \,]\!](\tau) = \begin{cases} I[\![\, e_2 \,]\!](\tau) & \text{if } I[\![\, e_1 \,]\!](\tau) = \text{true}, \\ I[\![\, e_3 \,]\!](\tau) & \text{if } I[\![\, e_1 \,]\!](\tau) = \text{false}, \\ \bot & \text{otherwise.} \end{cases}$$

v. $$I[\![\, (e \text{ asType } t') \,]\!](\tau) = \begin{cases} I[\![\, e \,]\!](\tau) & \text{if } I[\![\, e \,]\!](\tau) \in I(t'), \\ \bot & \text{otherwise.} \end{cases}$$

$$I[\![\, (e \text{ isTypeOf } t') \,]\!](\tau) = \begin{cases} \text{true} & \text{if } I[\![\, e \,]\!](\tau) \in I(t') - \bigcup_{t'' < t'} I(t''), \\ \text{false} & \text{otherwise.} \end{cases}$$

$$I[\![\, (e \text{ isKindOf } t') \,]\!](\tau) = \begin{cases} \text{true} & \text{if } I[\![\, e \,]\!](\tau) \in I(t'), \\ \text{false} & \text{otherwise.} \end{cases}$$

vi. $I[\![\, e_1 \to \text{iterate}(v_1; v_2 = e_2 \mid e_3) \,]\!](\tau) = I[\![\, e_1 \to \text{iterate}'(v_1 \mid e_3) \,]\!](\tau')$ where $\tau' = (\sigma, \beta')$ and $\tau'' = (\sigma, \beta'')$ are environments with modified variable assignments

$$\beta' := \beta\{v_2/I[\![\, e_2 \,]\!](\tau)\}$$
$$\beta'' := \beta'\{v_2/I[\![\, e_3 \,]\!](\sigma, \beta'\{v_1/x_1\})\}$$

and iterate$'$ is defined as:

(a) If $e_1 \in \text{Expr}_{Sequence(t_1)}$ then $I[\![e_1 \rightarrow \text{iterate}'(v_1 \mid e_3)]\!](\tau') =$

$$\begin{cases} I[\![v_2]\!](\tau') \\ \quad \text{if } I[\![e_1]\!](\tau') = \langle\rangle, \\ I[\![\text{mkSequence}_{t_1}(x_2, \ldots, x_n) \rightarrow \text{iterate}'(v_1 \mid e_3)]\!](\tau'') \\ \quad \text{if } I[\![e_1]\!](\tau') = \langle x_1, \ldots, x_n \rangle. \end{cases}$$

(b) If $e_1 \in \text{Expr}_{Set(t_1)}$ then $I[\![e_1 \rightarrow \text{iterate}'(v_1 \mid e_3)]\!](\tau') =$

$$\begin{cases} I[\![v_2]\!](\tau') \\ \quad \text{if } I[\![e_1]\!](\tau') = \emptyset, \\ I[\![\text{mkSet}_{t_1}(x_2, \ldots, x_n) \rightarrow \text{iterate}'(v_1 \mid e_3)]\!](\tau'') \\ \quad \text{if } I[\![e_1]\!](\tau') = \{x_1, \ldots, x_n\}. \end{cases}$$

(c) If $e_1 \in \text{Expr}_{Bag(t_1)}$ then $I[\![e_1 \rightarrow \text{iterate}'(v_1 \mid e_3)]\!](\tau') =$

$$\begin{cases} I[\![v_2]\!](\tau') \\ \quad \text{if } I[\![e_1]\!](\tau') = \emptyset, \\ I[\![\text{mkBag}_{t_1}(x_2, \ldots, x_n) \rightarrow \text{iterate}'(v_1 \mid e_3)]\!](\tau'') \\ \quad \text{if } I[\![e_1]\!](\tau') = \{\!\{x_1, \ldots, x_n\}\!\}. \end{cases}$$

□

The semantics of a variable expression (i) is the value assigned to the variable. A let expression (ii) results in the value of the sub-expression e_2. Free occurrences of the variable v in e_2 are bound to the value of the expression e_1. An operation expression (iii) is interpreted by the function associated with the operation. Each argument expression is evaluated separately. The state σ is passed to operations whose interpretation depends on the system state. These include, for example, attribute and navigation operations as defined in Section 4.4.

The computation of side effect-free operations can often be described with OCL expressions. We can extend the definition to allow object operations whose effects are defined in terms of OCL expressions. The semantics of a side effect-free operation can then be given by the semantics of the OCL expression associated with the operation. Recall that object operations in OP_c are declared in a model specification. Let $\text{oclexp} : \text{OP}_c \rightarrow \text{Expr}$ be a partial function mapping object operations to OCL expressions. We define the semantics of an operation with an associated OCL expression as

$$I[\![\omega(p_1 : e_1, \ldots, p_n : e_n)]\!](\tau) = I[\![\text{oclexp}(\omega)]\!](\tau')$$

where $p_1, \ldots, p_n$ are parameter names, and $\tau' = (\sigma, \beta')$ denotes an environment with a modified variable assignment defined as

$$\beta' := \beta\{p_1/I[\![e_1]\!](\tau), \ldots, p_n/I[\![e_n]\!](\tau)\} \ .$$

Argument expressions are evaluated and assigned to parameters that bind free occurrences of $p_1, \ldots, p_n$ in the expression oclexp(ω). For a well-defined semantics, we need to make sure that there is no infinite recursion resulting from an expansion of the operation call. A strict solution that can be statically checked is to forbid any occurrences of ω in oclexp(ω). However, allowing recursive operation calls considerably adds to the expressiveness of OCL (see the discussion in Section 5.3). We therefore allow recursive invocations as long as the recursion is finite. Unfortunately, this property is generally undecidable.

The result of an if-expression (iv) is given by the then-part if the condition is true. If the condition is false, the else-part is the result of the expression. An undefined condition makes the whole expression undefined. Note that when an expression in one of the alternative branches is undefined, the whole expression may still have a well-defined result. For example, the result of

```
if true then 1 else 1 div 0 endif
```

is 1. In OCL, the result would be undefined according to the general rule for undefined expressions since an undefined value appears as a sub-expression (see also the discussion on error handling on page 59).

The result of a cast expression (v) using asType is the value of the expression, if the value lies within the domain of the specified target type, otherwise it is undefined. A type test expression with isTypeOf is true if the expression value lies exactly within the domain of the specified target type without considering subtypes. An isKindOf type test expression is true if the expression value lies within the domain of the specified target type or one of its subtypes. Note that these type cast and test expressions also work with undefined values since every value – including an undefined one – has a well-defined type.

An iterate expression (vi) loops over the elements of a collection and allows the application of a function to each collection element. The function results are successively combined into a value that serves as the result of the whole iterate expression. This kind of evaluation is also known in functional style programming languages as *fold* operation (see, e.g., [Tho99]).

Figure 5.1 presents an algorithm for the evaluation of iterate expressions. We call the expression e_1 that provides a collection the *source expression*. The *argument expression* e_3 is then evaluated for each element in the collection. The argument expression may refer to the current element of the collection by using the *iteration variable* v_1 which is bound to a different collection element each time e_3 is evaluated. In order to accumulate the results of the argument expression, another variable v_2 is bound to the result of e_3 after each loop. This *accumulator variable* v_2 is initialized once with

the result of expression e_2 before looping over the elements of the source collection. To combine the current argument result with the previous one, the accumulator variable may be referenced in the argument expression. Finally, the result of the whole iterate expression is defined to be the final value of the accumulator v_2.

Algorithm for evaluating $e_1 \rightarrow \text{iterate}(v_1; v_2 \leftarrow e_2 \mid e_3)$:

$v_2 := e_2$
foreach v_1 **in** e_1 **do**
  $v_2 := e_3$
return v_2

1. Evaluate e_2 and bind the result to variable v_2.
2. Iterate over all elements in the collection e_1, and for each iteration, bind the variable v_1 to the current element.
3. Evaluate e_3 and bind the result to variable v_2. The expression e_3 may have free occurrences of v_1 and v_2.
4. The result is the final value of v_2.

Figure 5.1: Algorithm for evaluating iterate expressions

In Definition 5.2, the semantics of iterate expressions is given by a recursive evaluation scheme. Information is passed between different levels of recursion by modifying the variable assignment β appropriately in each step. The interpretation of iterate starts with the initialization of the accumulator variable. The recursive evaluation following thereafter uses a simplified version of iterate, namely an expression iterate$'$ where the initialization of the accumulator variable is left out, since this sub-expression needs to be evaluated only once. If the source collection is not empty, (1) an element from the collection is bound to the iteration variable, (2) the argument expression is evaluated, and (3) the result is bound to the accumulator variable. These steps are all part of the definition of the variable assignment β''. The recursion terminates when there are no more elements in the collection to iterate over. The constructor operations $\text{mkSequence}_t, \text{mkBag}_t$, and mkSet_t (see page 68) are in $\Omega_{\mathcal{M}}$ and provide the abstract syntax for collection literals like `Set{1,2}` in concrete OCL syntax.

One must be careful about what kind of expressions should be allowed as argument expressions. There may be expressions where the result of the whole iterate expression depends on the order in which collection elements are selected for application. 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 employees working at a car rental branch. We could use the following OCL expression for this purpose.

```
context b : Branch:
  b.employee->iterate(e : Employee;
    names : String = '' |
    names.concat(e.lastname))
```

The expression iterates over the elements in a set of employees (determined by the navigation expression `b.employee`) and adds the last name of each employee in this set to an initially empty string. The result will be a string containing all last names of the employees working at the given branch. The problem here is that there is no statement in OCL about the order in which elements from the set `b.employee` are selected for applying the argument expression. Hence, evaluations may yield different results caused by different iteration sequences. This shows that an important aspect of the iterate expression is under-specified in OCL. Since most operations on collections are defined in terms of iterate expressions their behavior is also not precisely defined.

To illustrate a solution to this problem, we reformulate the semantics of iterate expressions. Let C be a collection – a set, sequence, or a bag – with elements $x_1, \ldots, x_n$, and let x_0 be the initial value of the accumulator variable. The effect of the argument expression can be described by a binary operator $\oplus$ that combines the accumulator variable with some argument. Then the semantics of the iterate expression can be briefly stated as

$$(((x_0 \oplus x_1) \oplus x_2) \cdots \oplus x_{n-1}) \oplus x_n \ .$$

The order of elements is fixed only for sequences. For C being a set or a bag, 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 $(x_i \oplus x_{i+1}) \oplus x_{i+2} = (x_i \oplus x_{i+2}) \oplus x_{i+1}$ for $i = 0, \ldots, n-2$, that is, the ordering of elements does not matter. It can easily be seen that this property is fulfilled if $\oplus$ is associative and commutative. For operations where evaluation order indeed makes a difference (such as string concatenation), a non-ordered collection first has to be transformed into a sequence before iterate can be applied and a deterministic behavior is desired.

5.1.3 Derived Expressions Based on iterate

A number of important OCL constructs such as `exists`, `forAll`, `select`, `reject`, `collect`, and `isUnique` are defined in terms of iterate expressions. In [OMG99b], the intended semantics of these expressions is given by

postconditions with iterate-based expressions. The following schema shows how these expressions can be translated to equivalent iterate expressions. A similar translation can be found in [Cla99].

$$I[\![\, e_1 \rightarrow \text{exists}(v_1 \mid e_3) \,]\!](\tau) =$$
$$\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{false} \mid v_2 \text{ or } e_3) \,]\!](\tau)$$

$$I[\![\, e_1 \rightarrow \text{forAll}(v_1 \mid e_3) \,]\!](\tau) =$$
$$\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{true} \mid v_2 \text{ and } e_3) \,]\!](\tau)$$

$$I[\![\, e_1 \rightarrow \text{select}(v_1 \mid e_3) \,]\!](\tau) =$$
$$\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = e_1 \mid$$
$$\qquad \text{if } e_3 \text{ then } v_2 \text{ else } v_2 \rightarrow \text{excluding}(v_1) \text{ endif}) \,]\!](\tau)$$

$$I[\![\, e_1 \rightarrow \text{reject}(v_1 \mid e_3) \,]\!](\tau) =$$
$$\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = e_1 \mid$$
$$\qquad \text{if } e_3 \text{ then } v_2 \rightarrow \text{excluding}(v_1) \text{ else } v_2 \text{ endif}) \,]\!](\tau)$$

$$I[\![\, e_1 \rightarrow \text{collect}(v_1 \mid e_3) \,]\!](\tau) =$$
$$\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{mkBag}_{\textit{type-of-}e_3}() \mid v_2 \rightarrow \text{including}(e_3)) \,]\!](\tau)$$

$$I[\![\, e_1 \rightarrow \text{isUnique}(v_1 \mid e_3) \,]\!](\tau) =$$
$$\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{true} \mid v_2 \text{ and } e_1 \rightarrow \text{count}(v_1) = 1) \,]\!](\tau)$$

With these transformations, we are now able to exactly determine the result of quantifiers in presence of undefined values. For example, an exists expression is true if there is at least one element in e_1 making e_3 true. Any further undefined result for e_3 does not change the final outcome. Therefore, important questions regarding undefined values (which were first raised on page 24) can now be answered. In this case, Definition 5.2 provides the interpretation of iterate expressions, and Table 4.2 on page 58 gives the semantics of the boolean operation *or* which is used in the above translation of the exists quantifier.

5.1.4 Expression Context

An OCL expression is always written in some syntactical context. Since the primary purpose of OCL is the specification of constraints on a UML model, it is obvious that the model itself provides the most general kind of context. In our approach, the signature $\Sigma_{\mathcal{M}}$ contains types (e.g., object types) and operations (e.g., attribute operations) that are "imported" from a model, thus providing a context for building expressions that depend on the elements of a specific model.

On a much smaller scale, there is also a notion of context in OCL that simply introduces variable declarations. This notion is closely related to the syntax for constraints written in OCL. A context clause declares variables in invariants, and parameters in pre- and postconditions. The following example declares a variable `e` which is subsequently used in an invariant expression.

```
context e : Employee inv:
  e.age > 18
```

The next example declares a parameter `amount` which is used in a pre- and postcondition specification.

```
context Employee::raiseSalary(amount : Real) : Real
  pre:  amount > 0
  post: self.salary = self.salary@pre + amount
        and result = self.salary
```

Here we use the second meaning of context, that is, a context provides a set of variable declarations. The more general meaning of context is already subsumed by our concept of a signature as described above. A similar distinction between *local* and *global* declarations is also made in [CKM+99c]. In their paper, the authors extend the OCL context syntax to include global declarations and outline a general approach to derive declarations from information on the UML metamodel level.

A *context of an invariant* (corresponding to the nonterminal classifierContext in the OCL grammar [OMG99b]) is a declaration of variables. The variable declaration may be implicit or explicit. In the implicit form, the context is written as

```
context C inv:
  <expression>
```

In this case, the `<expression>` may use the variable `self` of type C as a free variable. In the explicit form, the context is written as

```
context v1 : C1, ..., vn : Cn inv:
  <expression>
```

The `<expression>` may use the variables $v_1, \ldots, v_n$ of types $C_1, \ldots, C_n$ as free variables. The OCL grammar actually only allows the explicit declaration of at most one variable in a classifierContext. This restriction seems unnecessarily strict. Having multiple variables is especially useful for constraints specifying key properties of attributes. The example (taken from [OMG99b, p. 7-18])

```
context Person inv:
  Person.allInstances->forAll(p1, p2 |
    p1 <> p2 implies p1.name <> p2.name)
```

could then be just written as:

```
context p1, p2 : Person inv:
  p1 <> p2 implies p1.name <> p2.name
```

A *context of a pre-/postcondition* (corresponding to the nonterminal `operationContext` in the OCL grammar) is a declaration of variables. In this case, the context is written as

context $C :: \mathtt{op}(p_1 : T_1, \ldots, p_n : T_n) : T$
pre: P
post: Q

This means that the variable `self` (of type C) and the parameters $p_1, \ldots, p_n$ may be used as free variables in the precondition P and the postcondition Q. Additionally, the postcondition may use `result` (of type T) as a free variable. The details are explained in Section 5.2.

5.1.5 Invariants

An invariant is an expression with boolean result type and a set of (explicitly or implicitly declared) free variables $v_1 : C_1, \ldots, v_n : C_n$ where $C_1, \ldots, C_n$ are classifier types. An invariant

context $v_1 : C_1, \ldots, v_n : C_n$ **inv:**
  <expression>

is equivalent to the following expression without free variables that must be valid in all system states.

```
C1.allInstances->forAll(v1 : C1 |
   ...
   Cn.allInstances->forAll(vn : Cn |
     <expression>
   )
   ...
)
```

A system state is called valid with respect to an invariant if the invariant evaluates to true. Invariants with undefined result invalidate a system state. The following examples all specify invariants of the class *Employee*.

```
-- all employees have a non-empty attribute lastname
context Employee inv:
  self.lastname <> ''

-- same as above but using a variable instead of self
context e : Employee inv:
  e.lastname <> ''

-- last names are unique among all employees
context e1, e2 : Employee inv:
  e1 <> e2 implies e1.lastname <> e2.lastname
```

In this example, the context of an invariant is given by a class name and a variable list. If no variable is declared, the *self* keyword can be used to refer to an object of the context class. Note that although most invariants are attached to a single class, an invariant may still refer to objects of different classes by means of navigation. The invariants given above can be rewritten as equivalent but fully self-contained expressions by explicitly specifying the universal quantifier implied by an invariant. This example also shows that a special concept for "self" is not necessary since it can simply be treated as a variable.

```
Employee.allInstances->forAll(self : Employee |
  self.lastname <> '')

Employee.allInstances->forAll(e : Employee |
  e.lastname <> '')

Employee.allInstances->forAll(e1 : Employee |
  Employee.allInstances->forAll(e2 : Employee |
    e1 <> e2 implies e1.lastname <> e2.lastname))
```

Extensions

Global invariants could be very useful for conditions that must hold for all objects of *all* classes, i.e., they must be true for any system state. Attaching this kind of invariant to an arbitrary class is not equivalent since the expression is always true when there are no objects of this class.

In the previous chapter, we have seen that association types nicely complement object types. In the same way in which we attach invariants to classes,

we can also add invariants to associations. In the following example, the variable `self` is bound to links of the association R which is assumed to be a self association on a class A. The role names `r1` and `r2` refer to objects of A connected by a single link.

```
-- links of the self association R
-- must connect different objects
context R inv:
  self.r1 <> self.r2
```

5.1.6 Queries

OCL has no concept of a query but the term has proven to be useful. We define a *query* to be an OCL expression with arbitrary result type and no free variables. A context declaration is not given. The following query selects from the set of all persons those whose name is Knuth.

```
Person.allInstances->select(p | p.name = 'Knuth')
```

5.1.7 Shorthand Notations

There are a few shorthand notations in OCL which we can define by simple syntactical transformations. A related approach without giving technical details is described in [HDF00] where a "normalization" step transforms various kinds of OCL expressions into a normal form preserving invariants specified on the abstract syntax tree.

Shorthand for `collect`

Navigation operations followed by a `collect` expression frequently occur in OCL. These expressions are commonly used to retrieve some property for all objects connected to a given source object. The resulting collection is automatically flattened. Let `coll` be an expression with a collection type $Collection(C)$, $Set(C)$, $Bag(C)$, or $Sequence(C)$ where C denotes an object type. If `op` is an operation defined on objects of type C then the expression

```
coll.op()
```

is a shorthand notation which is equivalent to the expression

```
coll->collect(o : C | o.op())
```

If this expression results in a nested collection, we additionally apply a flattening step to preserve the intended OCL meaning:

```
coll->collect(o : C | o.op())->flatten()
```

The parentheses after `op` are omitted in the concrete syntax if the operation denotes an attribute access or a navigation by role name operation. In contrast to the definition of the shorthand notation in OCL, the above translation also makes the flattening process explicit. In the following example, we navigate only one association (see the class diagram in Figure 2.2 on page 11). The expression

```
branch.employee.lastname
```

collects the last names of all employees of a given branch. The translation and a possible result are as follows.

```
branch.employee->collect(e | e.lastname)
-- Bag{'Green','White'} : Bag(String)
```

In the next example, we navigate along two associations resulting in a nested collection. The expression

```
branch.car.check
```

collects all maintenance checks for all cars of a given branch. This time, the translation adds a flattening step, because otherwise the result type would be $Bag(Set(Check))$.

```
branch.car->collect(c : Car | c.check)->flatten()
-- Bag{@check1} : Bag(Check)
```

In the result, we use the @-symbol to refer to an object since OCL does not have a notation for object literals. For illustrating the effect of flattening, we also give the expression without the flatten operation. The result value corresponds to the same system state as the example above.

```
branch.car->collect(c : Car | c.check)
-- Bag{Set{},Set{},Set{},Set{@check1}} : Bag(Set(Check))
```

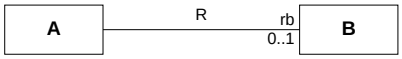


Figure 5.2: Binary association with multiplicity `0..1`

Navigation over associations with multiplicity zero or one

An association end with a multiplicity of `0..1` induces a navigation operation with the result type equal to the object type of the class at the target end. For the situation shown in Figure 5.2 this means that there is a navigation operation $rb : A \rightarrow B$. The result of the operation is either an object of type B that is linked to a given A object, or undefined if no such object exists.

OCL defines a shorthand notation to treat the result of an expression `a.rb` as a set [OMG99b, p. 7-13]. For example, the expression `a.rb->size` yields 1 if a B object is connected to `a` and 0 otherwise. Since our approach explicitly supports undefined values we can simply map the OCL shorthand notation to the following expression.

```
if a.rb.isUndefined() then
  Set{}
else
  Set{a.rb}
endif
```

This mapping applies whenever an expression `a.rb` referring to an association end with multiplicity zero or one is used in a set context, i.e., it is the source of a set operation.

Multiple iterator variables

It is sometimes convenient to integrate multiple iterations over a collection into a single iterate expression. For example, in order to compute

$$\begin{aligned}\sum_{1\leq i,j\leq 3}(i*j) = \quad & (1*1)+(2*1)+(3*1)\\ + \quad & (1*2)+(2*2)+(3*2)\\ + \quad & (1*3)+(2*3)+(3*3)\end{aligned}$$

a corresponding OCL expression requires two nested iterate expressions:

```
Sequence{1,2,3}->iterate(e1 : Integer;
   res1 : Integer = 0 |
   res1 + Sequence{1,2,3}->iterate(e2 : Integer;
     res2 : Integer = 0 |
     res2 + e1 * e2))
```

An equivalent expression with only one iterate expression can be given if we allow multiple iterator variables. Each variable `e1` and `e2` iterates over the whole collection.

```
Sequence{1,2,3}->iterate(
     e1, e2 : Integer;
     res : Integer = 0 |
     res + e1 * e2)
```

Although this feature is not explained for iterate expressions in [OMG99b], examples can be found for the iterate-based forAll construct where two variables are used [OMG99b, p. 7-18]. Since many constructs are mapped to iterate expressions it makes sense to introduce the above shorthand for the generic iterate expression.

5.2 Pre- and Postconditions

The definition of expressions in the previous section is sufficient for invariants and queries where we have to consider only single system states. For pre- and postconditions, there are additional language constructs in OCL which enable references to the system state before the execution of an operation and to the system state that results from the operation execution. The general syntax of an operation specification with pre- and postconditions is defined as

context $C :: \texttt{op}(p_1 : T_1, \ldots, p_n : T_n)$
  pre: P
  post: Q

First, the context is determined by giving the signature of the operation for which pre- and postconditions are to be specified. The operation `op` which is defined as part of the classifier C has a set of typed parameters $\text{PARAMS}_{\text{op}} = \{p_1, \ldots, p_n\}$. The UML model providing the definition of an operation signature also specifies the direction kind of each parameter. We use a function $kind : \text{PARAMS}_{\text{op}} \rightarrow \{\text{in}, \text{out}, \text{inout}, \text{return}\}$ to map each parameter to one of these kinds. Although UML makes no restriction on the number of return parameters, there is usually only at most one return parameter considered in OCL which is referred to by the keyword `result` in a postcondition. In this case, the signature is also written as $C :: \texttt{op}(p_1 : T_1, \ldots, p_{n-1} : T_{n-1}) : T$ with T being the type of the `result` parameter.

The precondition of the operation is given by an expression P, and the postcondition is specified by an expression Q. P and Q must have a boolean

result type. If the precondition holds, the contract of the operation guarantees that the postcondition is satisfied after completion of `op`. Pre- and postconditions form a pair. A condition defaults to true if it is not explicitly specified.

Note that in previous sections, we have talked about side effect-free operations. Now we are discussing operations that usually have side effects. Table 5.1 summarizes different kinds of operations in UML. Operations in the table are classified by the existence of a return parameter in the signature, whether they are declared as being side effect-free (with the tag *isQuery* in UML), the state before and after execution, and the languages in which (1) the operation body can be expressed (Body), and (2) the operation may be called (Caller).

Return value	side effect-free	States	Body	Caller
–	–	pre-state ≠ post-state allowed	AL	AL
•	–	pre-state ≠ post-state allowed	AL	AL
•	•	pre-state = post-state required	OCL	OCL, AL

Table 5.1: Different kinds of operations in UML

The first row of the table describes operations without a return value. These are used to specify side effects on a system state. Therefore, the post-state usually differs from the state before the operation call. Since specifying side effects is out of the scope of OCL expressions, the body of the operation must be expressed in some kind of *Action Language* (AL). Furthermore, the operation cannot be used without restriction as part of an OCL expression because all operations in an OCL expression must be tagged *isQuery*. The same arguments apply to operations with a return value that are listed in the second row. The third kind of operations are those operations which may be used in OCL without restrictions. Because their execution does not have side effects, the pre- and post-states are always equal. Often, the body of the operation can be specified with an OCL expression. It might be desirable for an action language to make use of these kinds of operations by including OCL as a sub-language.

5.2.1 Motivating Example

Before we give a formal definition of operation specifications with pre- and postconditions, we demonstrate the fundamental concepts by means of an example. Figure 5.3 shows a class diagram with two classes A and B that are related to each other by an association R. Class A has an operation `op()` but no attributes. Class B has an attribute c and no operations. The implicit

role names `a` and `b` at the link ends allow navigation in OCL expressions from a B object to the associated A object and vice versa.

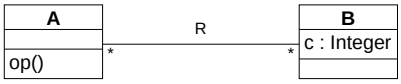
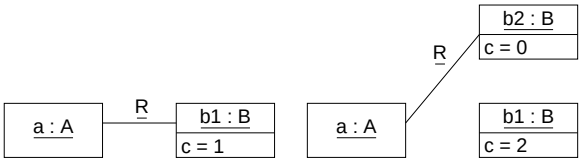


Figure 5.3: Example class diagram

Figure 5.4 shows an example for two consecutive states of a system corresponding to the given class model. The object diagrams show instances of classes A and B and links of the association R. The left object diagram shows the state before the execution of an operation, whereas the right diagram shows the state after the operation has been executed. The effect of the operation can be described by the following changes in the post-state: (1) the value of the attribute c in object $\underline{b}_1$ has been incremented by one, (2) a new object $\underline{b}_2$ has been created, (3) the link between $\underline{a}$ and $\underline{b}_1$ has been removed, and (4) a new link between $\underline{a}$ and $\underline{b}_2$ has been established.



(a) Pre-state with objects $\underline{a}$ and $\underline{b}_1$.

(b) Post-state. Object $\underline{b}_2$ did not exist in the pre-state.

Figure 5.4: Object diagrams showing a pre- and a post-state

For the following discussion, consider the OCL expression `a.b.c` where `a` is a variable denoting the object $\underline{a}$. The expression navigates to the associated object of class B and results in the value of the attribute c. Therefore, the expression evaluates to 1 in the pre-state shown in Figure 5.4(a). As an example of how the OCL modifier `@pre` may be used in a postcondition to refer to properties of the previous state, we now look at some variations of the expression `a.b.c` that may appear as part of a postcondition. For each case, the result is given and explained.

- `a.b.c = 0`
  Because the expression is completely evaluated in the post-state, the navigation from $\underline{a}$ leads to the $\underline{b}_2$ object. The value of the attribute c of $\underline{b}_2$ is 0 in Figure 5.4(b).

- `a.b@pre.c = 2`
  This expression refers to both the pre- and the post-state. The previous value of `a.b` is a reference to object $\underline{b}_1$. However, since the `@pre` modifier only applies to the expression `a.b`, the following reference to the attribute c is evaluated in the post-state of $\underline{b}_1$, even though $\underline{b}_1$ is not connected anymore to $\underline{a}$. Therefore, the result is 2.

- `a.b@pre.c@pre = 1`
  In this case, the value of the attribute c of object $\underline{b}_1$ is taken from the pre-state. This expression is semantically equivalent to the expression `a.b.c` in a precondition.

- `a.b.c@pre = ⊥`
  The expression `a.b` evaluated in the post-state yields a reference to object $\underline{b}_2$ which is now connected to $\underline{a}$. Since $\underline{b}_2$ has just been created by the operation, there is no previous state of $\underline{b}_2$. Hence, a reference to the previous value of attribute c is undefined.

Note that the `@pre` modifier may only be applied to operations not to arbitrary expressions. An expression such as `(a.b)@pre` is syntactically illegal.

OCL provides the standard operation `oclIsNew` for checking whether an object has been created during the execution of an operation. This operation may only be used in postconditions. For our example, the following conditions indicate that the object $\underline{b}_2$ has just been created in the post-state and $\underline{b}_1$ already existed in the pre-state.

- `a.b.oclIsNew = true`
- `a.b@pre.oclIsNew = false`

5.2.2 Syntax and Semantics of Postconditions

All common OCL expressions can be used in a precondition P. Syntax and semantics of preconditions are defined exactly like those for plain OCL expressions in Section 5.1. Also, all common OCL expressions can be used in a postcondition Q. Additionally, the `@pre` construct, the special variable `result`, and the operation `oclIsNew` may appear in a postcondition. In the following, we extend Definition 5.1 for the syntax of OCL expressions to provide these additional features.

Definition 5.3 (Syntax of expressions in postconditions)
Let op be an operation with a set of parameters $\text{PARAMS}_{\text{op}}$. The set of parameters includes at most one parameter of kind "return". The basic

set of expressions in postconditions is defined by repeating Definition 5.1 while substituting all occurrences of Expr_t with Post-Expr_t. Furthermore, we define that

- Each non-return parameter $p \in \text{PARAMS}_{\text{op}}$ with a declared type t is available as variable: $p \in \text{Var}_t$.
- If $\text{PARAMS}_{\text{op}}$ contains a parameter of kind "return" and type t then `result` is a variable: $\texttt{result} \in \text{Var}_t$.
- The operation $\texttt{oclIsNew} : c \to \mathit{Boolean}$ is in $\Omega_{\mathcal{M}}$ for all object types $c \in T_{\mathcal{M}}$.

The syntax of expressions in postconditions is extended by the following rule.

vii. If $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_{\mathcal{M}}$ and $e_i \in \text{Post-Expr}_{t'}$ for all $i = 1, \ldots, n$ then $\boldsymbol{\omega}_{@\text{pre}}(\boldsymbol{e_1}, \ldots, \boldsymbol{e_n}) \in \text{Post-Expr}_t$.

□

All general OCL expressions may be used in a postcondition. Moreover, the basic rules for recursively constructing expressions do also apply. Operation parameters are added to the set of variables. For operations with a return type, the variable `result` refers to the operation result. The set of operations is extended by `oclIsNew` which is defined for all object types. Operations $\omega_{@\text{pre}}$ are added for allowing references to the previous state (vii). The rule says that the `@pre` modifier may be applied to all operations, although, in general, not all operations do actually depend on a system state (for example, operations on data types). The result of these operations will be the same in all states. Operations which do depend on a system state are, e.g., attribute access and navigation operations.

For a definition of the semantics of postconditions, we will refer to *environments* describing the previous state and the state resulting from executing the operation. An environment $\tau = (\sigma, \beta)$ is a pair consisting of a system state σ and a variable assignment β (see Section 5.1.2). The necessity of including variable assignments into environments will be discussed shortly. We call an environment $\tau_{\text{pre}} = (\sigma_{\text{pre}}, \beta_{\text{pre}})$ describing a system state and variable assignments before the execution of an operation a *pre-environment*. Likewise, an environment $\tau_{\text{post}} = (\sigma_{\text{post}}, \beta_{\text{post}})$ after the completion of an operation is called a *post-environment*.

Definition 5.4 (Semantics of postcondition expressions)
Let Env be the set of environments. The semantics of an expression $e \in \text{Post-Expr}_t$ is a function $I[\![e]\!] : \text{Env} \times \text{Env} \to I(t)$. The semantics of the basic set of expressions in postconditions is defined by repeating Definition 5.2 while substituting all occurrences of Expr_t with Post-Expr_t. References to $I[\![e]\!](\tau)$ are replaced by $I[\![e]\!](\tau_{\text{pre}}, \tau_{\text{post}})$ to include the pre-environment. Occurrences of τ are changed to τ_{post} which is the default environment in a postcondition.

- For all $p \in \text{PARAMS}_{\text{op}}$: $I[\![p]\!](\tau_{\text{pre}}, \tau_{\text{post}}) = \beta_{\text{post}}(p)$.
  - Input parameters may not be modified by an operation: $\text{kind}(p) = \text{in}$ implies $\beta_{\text{pre}}(p) = \beta_{\text{post}}(p)$.
  - Output parameters are undefined on entry: $\text{kind}(p) = \text{out}$ implies $\beta_{\text{pre}}(p) = \bot$.
- $I[\![\texttt{result}]\!](\tau_{\text{pre}}, \tau_{\text{post}}) = \beta_{\text{post}}(\texttt{result})$.
- $I[\![\texttt{oclIsNew}]\!](\tau_{\text{pre}}, \tau_{\text{post}})(\underline{c}) = \begin{cases} \text{true} & \text{if } \underline{c} \notin \sigma_{\text{pre}}(c), \\ \text{false} & \text{otherwise.} \end{cases}$

vii. $I[\![\omega_{@\text{pre}}(e_1, \ldots, e_n)]\!](\tau_{\text{pre}}, \tau_{\text{post}}) =$
$I(\omega)(\tau_{\text{pre}})(I[\![e_1]\!](\tau_{\text{pre}}, \tau_{\text{post}}), \ldots, I[\![e_n]\!](\tau_{\text{pre}}, \tau_{\text{post}}))$

□

Standard expressions are evaluated as defined in Definition 5.2 with the post-environment determining the context of evaluation. Input parameters do not change during the execution of the operation. Therefore, their values are equal in the pre- and post-environment. The value of the `result` variable is determined by the variable assignment of the post-environment. The `oclIsNew` operation yields true if an object did not exist in the previous system state. Operations referring to the previous state are evaluated in context of the pre-environment (vii). Note that the operation arguments may still be evaluated in the post-environment. Therefore, in a nested expression, the environment only applies to the current operation, whereas deeper nested operations may evaluate in a different environment.

With these preparations, the semantics of an operation specification with pre- and postconditions can be precisely defined as follows. We say that a precondition P *satisfies* a pre-environment τ_{pre} – written as $\tau_{\text{pre}} \models P$ – if the expression P evaluates to true according to Definition 5.2. Similarly, a postcondition Q satisfies a pair of pre- and post-environments, if the

expression Q evaluates to true according to Definition 5.4:

$$\begin{aligned} \tau_{\text{pre}} \models P \quad &\text{iff} \quad I[\![P]\!](\tau_{\text{pre}}) = \text{true} \\ (\tau_{\text{pre}}, \tau_{\text{post}}) \models Q \quad &\text{iff} \quad I[\![Q]\!](\tau_{\text{pre}}, \tau_{\text{post}}) = \text{true} \end{aligned}$$

Definition 5.5 (Semantics of operation specifications)
The semantics of an operation specification is a set $R \subseteq \text{Env} \times \text{Env}$ defined as

$$\begin{array}{l} [\![\ \textbf{context}\ C :: \mathsf{op}(p_1 : T_1, \ldots, p_n : T_n) \\ \quad \textbf{pre:}\ P \\ \quad \textbf{post:}\ Q \ ]\!] \ = \ R \end{array}$$

where R is the set of all pre- and post-environment pairs such that the pre-environment τ_{pre} satisfies the precondition P and the pair of both environments satisfies the postcondition Q:

$$R = \{(\tau_{\text{pre}}, \tau_{\text{post}}) \mid \tau_{\text{pre}} \models P \ \wedge \ (\tau_{\text{pre}}, \tau_{\text{post}}) \models Q\}$$

□

The satisfaction relation for Q is defined in terms of both environments since the postcondition may contain references to the previous state. The set R defines all legal transitions between two states corresponding to the effect of an operation. It therefore provides a framework for a correct implementation.

Definition 5.6 (Satisfaction of operation specifications)
An operation specification with pre- and postconditions is satisfied by a program S in the sense of total correctness if the computation of S is a total function $f_S : \text{dom}(R) \rightarrow \text{im}(R)$ and $\text{graph}(f_S) \subseteq R$. □

In other words, the program S accepts each environment satisfying the precondition as input and produces an environment that satisfies the postcondition. The definition of R allows us to make some statements about the specification. In general, a reasonable specification implies a non-empty set R allowing one or more different implementations of the operation. If $R = \emptyset$, then there is obviously no implementation possible. We distinguish two cases: (1) no environment satisfying the precondition exists, or (2) there are environments making the precondition true, but no environments do satisfy the postcondition. Both cases indicate that the specification is inconsistent with the model. Either the constraint or the model providing the context should be changed. A more restrictive definition might even prohibit the second case.

5.2.3 Examples

Example 1. Consider the operation `raiseSalary` which raises the salary of an employee by a certain amount and returns the new salary.

```
context Employee::raiseSalary(amount : Real) : Real
  pre:  amount > 0
  post: result = self.salary
  post: self.salary = self.salary@pre + amount
```

The precondition only allows positive values for the amount parameter. The postcondition is specified as two parts which must both be true after executing the operation. This could equivalently be rephrased into a single expression combining both parts with a logical *and*. The first postcondition specifies that the result of the operation must be equal to the salary in the post-state. The second postcondition defines the new salary to be equal to the sum of the old salary and the amount parameter. All system states making the postcondition true after a call to `raiseSalary` has completed, satisfy the operation specification.

Example 2. The above example gives an exclusive specification of the operation's effect. The result is uniquely defined by the postconditions. Compare this with the next example giving a much looser specification of the result.

```
context Employee::raiseSalary(amount : Real) : Real
  pre:  amount > 0
  post: result > self.salary@pre
```

The result may be any value greater than the value of the salary in the previous state. Thus, the postcondition does not even prevent the salary from being decreased. However, what the example should make clear, is that there may not only exist many post-states but also many bindings of the result variable satisfying a postcondition. This is the reason why we have to consider both the system state *and* the set of variable bindings for determining the environment of an expression in a postcondition.

Example 3. This example shows the evaluation of some expressions that may appear in a postcondition. An informal explanation of the expressions was given in Section 5.2.1. With the previous syntax and semantics definitions, we are now able to give a precise meaning to these expressions. Roman numbers in parentheses at the right of the transformations show which rule of Definition 5.4 (see also Definition 5.2) has been applied in each step.

1. `a.b.c`

$$\begin{aligned}
I[\![\, c(b(a)) \,]\!](\tau_{\text{pre}}, \tau_{\text{post}}) &= I(c)(\tau_{\text{post}})(I[\![\, b(a) \,]\!](\tau_{\text{pre}}, \tau_{\text{post}})) && \text{(iii)}\\
&= I(c)(\tau_{\text{post}})(I(b)(\tau_{\text{post}})(I[\![\, a \,]\!](\tau_{\text{pre}}, \tau_{\text{post}}))) && \text{(iii)}\\
&= I(c)(\tau_{\text{post}})(I(b)(\tau_{\text{post}})(\beta(a))) && \text{(i)}\\
&= I(c)(\tau_{\text{post}})(I(b)(\tau_{\text{post}})(\underline{a}))\\
&= I(c)(\tau_{\text{post}})(\underline{b}_2)\\
&= 0
\end{aligned}$$

2. `a.b@pre.c`

$$\begin{aligned}
I[\![\, c(b_{@\text{pre}}(a)) \,]\!](\tau_{\text{pre}}, \tau_{\text{post}}) &= I(c)(\tau_{\text{post}})(I[\![\, b_{@\text{pre}}(a) \,]\!](\tau_{\text{pre}}, \tau_{\text{post}})) && \text{(iii)}\\
&= I(c)(\tau_{\text{post}})(I(b)(\tau_{\text{pre}})(I[\![\, a \,]\!](\tau_{\text{pre}}, \tau_{\text{post}}))) && \text{(vii)}\\
&= I(c)(\tau_{\text{post}})(I(b)(\tau_{\text{pre}})(\beta(a))) && \text{(i)}\\
&= I(c)(\tau_{\text{post}})(I(b)(\tau_{\text{pre}})(\underline{a}))\\
&= I(c)(\tau_{\text{post}})(\underline{b}_1)\\
&= 2
\end{aligned}$$

3. `a.b@pre.c@pre`

$$\begin{aligned}
I[\![\, c_{@\text{pre}}(b_{@\text{pre}}(a)) \,]\!](\tau_{\text{pre}}, \tau_{\text{post}}) &= I(c)(\tau_{\text{pre}})(I[\![\, b_{@\text{pre}}(a) \,]\!](\tau_{\text{pre}}, \tau_{\text{post}})) && \text{(vii)}\\
&= I(c)(\tau_{\text{pre}})(I(b)(\tau_{\text{pre}})(I[\![\, a \,]\!](\tau_{\text{pre}}, \tau_{\text{post}}))) && \text{(vii)}\\
&= I(c)(\tau_{\text{pre}})(I(b)(\tau_{\text{pre}})(\beta(a))) && \text{(i)}\\
&= I(c)(\tau_{\text{pre}})(I(b)(\tau_{\text{pre}})(\underline{a}))\\
&= I(c)(\tau_{\text{pre}})(\underline{b}_1)\\
&= 1
\end{aligned}$$

4. `a.b.c@pre`

$$\begin{aligned}
I[\![\, c_{@\text{pre}}(b(a)) \,]\!](\tau_{\text{pre}}, \tau_{\text{post}}) &= I(c)(\tau_{\text{pre}})(I[\![\, b(a) \,]\!](\tau_{\text{pre}}, \tau_{\text{post}})) && \text{(vii)}\\
&= I(c)(\tau_{\text{pre}})(I(b)(\tau_{\text{post}})(I[\![\, a \,]\!](\tau_{\text{pre}}, \tau_{\text{post}}))) && \text{(iii)}\\
&= I(c)(\tau_{\text{pre}})(I(b)(\tau_{\text{post}})(\beta(a))) && \text{(i)}\\
&= I(c)(\tau_{\text{pre}})(I(b)(\tau_{\text{post}})(\underline{a}))\\
&= I(c)(\tau_{\text{pre}})(\underline{b}_2)\\
&= \bot
\end{aligned}$$

5.3 Expressiveness

In this section, we discuss the expressiveness of the Object Constraint Language. The main results were first introduced by Mandel and Cengarle

in [MC99]. They showed "that OCL is not powerful enough to denote any query expression of the relational calculus. However, by means of OCL it is possible to calculate the transitive closure of a relation". They also argued that primitive recursive functions can be calculated by OCL expressions but that OCL is not Turing complete.

Relational algebra/calculus

Mandel and Cengarle first investigated whether the five basic operations of the relational algebra (union, difference, Cartesian product, projection, and selection) can be expressed in OCL. This is a prerequisite for a language to be complete in the sense defined by Ullman [Ull82]. The following summarizes their results in [MC99].

- *Union* is available as a predefined operation on collections: `set1->union(set2)`
- *Difference* is a predefined operation on sets: `set1 - set2`
- *Cartesian product* is not directly expressible in OCL because new types can not be created "on the fly". On the other hand, a class AB simulating the Cartesian product of two other classes A and B can be added to a model.
- *Projection* is possible for single attributes using `collect` on collections: `collection->collect(attrName)`
- *Selection* is possible using `select` expressions on collections: `collection->select(<boolean-expression>)`

Other derived operations of the relational algebra like intersection, join, and natural join can also be expressed in OCL. Some operations such as intersection even have a direct counterpart in OCL. A major problem is the lack of a tuple type and nested collections. As a consequence, considerable effort is required, for example, to represent a relation.

The conclusion of the above observations is that OCL is incomplete with respect to the relational algebra. Completeness can be achieved by adding tuple types and the possibility for creating instances of any type [MC99]. In Chapter 4, we have shown that it is straightforward to extend the OCL type system with tuple types. This lightweight extension would close one important gap between OCL and relational query languages and thus would allow OCL to benefit from results known from relational languages.

Transitive closure

The relational algebra is not expressive enough to compute the transitive closure of a relation. However, this operation is useful, for example, to recursively determine all descendants of a person, all subparts of a hierarchically structured part of a production, or generally for determining cycles in aggregation hierarchies. Computing the transitive closure is possible in OCL because of the looping construct provided by the iterate expression. This allows to use the non-recursive algorithm by Warshall which is based on an adjacency matrix representation of a relation (see, e.g., [Sch97]).

In the following OCL expression, we use Warshall's algorithm to compute the transitive closure of a relation. In the example, the input is a set $M = \{1, 2, 3, 4\}$ and a relation $R \subseteq M \times M$ with $R = \{(1, 2), (2, 3), (2, 4)\}$. We express a binary relation as a set of sequences where each sequence represents a pair of elements in M. The result of the expression is a set $R^+ = Set\{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4)\}$.

```
Set{1,2,3,4}->iterate(x,y,z : Integer;
  r : Set(Sequence(Integer))
    = Set{Sequence{1,2},Sequence{2,3},Sequence{2,4}} |
  if r->exists(p1, p2 : Sequence(Integer) |
    p1->at(1) = x and p1->at(2) = y and
    p2->at(1) = y and p2->at(2) = z)
  then
    r->including(Sequence{x,z})
  else
    r
  endif)
```

We use three iterator variables x, y, z each ranging over all elements of M. A pair (x, z) is included in the resulting relation R^+ if it is already in R or if there is an y such that pairs (x, y) and (y, z) are also in R^+. Note that the example makes use of nested collections by representing relations as sets of sequences. This would not be possible in standard OCL which does not allow nested collections. In [MC99] an application of Warshall's algorithm with standard OCL is shown which is considerably more complex than our expression above (the expression takes one and a half page). This complexity results from a tricky encoding of the relation structure to circumvent the problem of automatic flattening.

Turing completeness

In [MC99], it is shown that LOOP-programs can be encoded in OCL. LOOP-programs represent the class of primitive recursive functions. Again,

the OCL iterate construct is the key feature here since loops with a previously known number of iterations can be expressed with it. On the other hand, WHILE-programs with a previously unknown number of iterations cannot be mapped to iterate expressions (or any other construct) in OCL. Hence, it is not possible to express μ-recursive functions in OCL. The language is Turing incomplete.

A way to increase the expressiveness of OCL is to add recursive operation invocations. UML already allows to tag operations in a model with an *isQuery* attribute specifying that an operation has no side effects. In this case, the operation can be "implemented" with an OCL expression. If the expression directly or indirectly refers to the containing operation by means of an operation call, we get the computational power of recursive functions. The price for the increased expressiveness is that, in general, we cannot guarantee anymore that the evaluation of an expression always terminates.

Summary

In this chapter, we defined the abstract syntax and semantics of OCL expressions. Together with Chapter 4 this provides a complete formalization of OCL expressions. We also defined the context of expressions and their usage in constraints such as invariants and pre- and postconditions. We briefly summarize our main contributions to an improved understanding of OCL.

- The OCL standard [OMG99b] does not say what exactly constitutes an expression and how an expression should be interpreted. This is made precise by Definitions 5.1 and 5.2.
- The possibility of non-determinism in some iterate-based expressions is made clear and criteria for achieving deterministic iterate expressions are given.
- Precise meaning was given to notions such as context, invariant, and query.
- We defined a small OCL core that can be extended by semantics preserving transformation rules. Examples are derived expressions such as forAll and select which can be build upon iterate. Furthermore, the meaning of shorthand notations found in many OCL applications is made precise by transformations of these constructs to the language core.
- We have shown that postconditions nicely fit into the framework of a core language. Only a few orthogonal extensions of the expression syn-

tax and semantics were necessary to support the additional language features available in postconditions.

Chapter 6

A Metamodel for OCL

The Object Constraint Language allows the extension of UML models with constraints in a formal way. While most of UML is defined by following a metamodeling approach, there is currently no equivalent definition for OCL. We propose a metamodel for OCL that fills this gap. The benefit of a metamodel for OCL is that it precisely defines the structure and syntax of all OCL concepts like types, expressions, and values in an abstract way and by means of UML features. Thus, all legal OCL expressions can be systematically derived and instantiated from the metamodel. We also show that our metamodel smoothly integrates with the UML metamodel. The focus of this work lies on the syntax of OCL – the metamodel does not include a definition of the semantics of constraints. A previous version of the work presented in this chapter has been published in [RG99a]. It has since then been used for various extensions and modifications, for example, in [BH00] and [Bod00].

This chapter is structured as follows. The general approach that we followed to define a metamodel for OCL is explained in Section 6.1. In Section 6.2, we illustrate the package structure of the OCL metamodel. Since OCL is primarily used for specifying constraints on UML models, there exists a strong relationship between both metamodels. A link between OCL and the UML core model is established in Section 6.3. The *Types* package of our metamodel is presented in Section 6.4. The central part of the metamodel is the definition of *Expressions* which are discussed in Section 6.5. The result of evaluating expressions are *Values* which are not strictly part of the abstract syntax but are included in Section 6.6 for completeness. In each section, examples are given that illustrate the application of the metamodel to concrete OCL types, expressions, and values.

6.1 General Approach

The main reference for UML is the OMG standard [OMG99c]. This document defines the language to be used for specifying well-formed UML models. Language constructs are defined by an abstract syntax, a set of well-formedness rules, and an informal description of the intended semantics. The abstract syntax is defined as a metamodel in form of UML class diagrams. This style of presentation has been chosen for most UML parts but not for OCL. In order to achieve a uniform presentation for the complete UML *including* OCL, we therefore propose a definition of OCL following the style of [OMG99e]. Our main contribution to achieve this goal is the definition of an OCL metamodel.

The metamodel presented in this chapter is not just a different kind of presentation of OCL. We rather consider it a complementary work to the OCL definition given in [OMG99b]. The OCL document concentrates on the concrete syntax and application aspects of the Object Constraint Language. Our metamodel, on the other hand, tries to define the abstract syntax of OCL. Another benefit resulting from having a metamodel is that it is quite easy to derive an implementation. For example, we have used the metamodel as a design for an OCL interpreter being part of a tool for validating UML models and OCL constraints. The tool is described in more detail in Chapter 7.

The UML metamodel is defined as one of the layers of a four-layer metamodeling architecture [OMG99e, p. 2-4]. In this architecture the UML metamodel is an instance of the meta-metamodel of the OMG *Meta Object Facility* (MOF) [OMG99a]. For an integration of the OCL metamodel with this architecture there are basically two options. We can place the OCL metamodel at the same level (the M2 level) as the UML metamodel and use the MOF as the definition language (provided by the M3 level). The result is a general MOF-compliant metamodel of OCL. The second option is to lift the OCL metamodel to the M3 level. One reason is that OCL may obviously apply to other metamodels than just UML. Furthermore, the MOF also uses OCL for specifying constraints at the M3 level. Consequently, it would make sense to define the OCL metamodel as part of the MOF. This way our work can be seen as a contribution to the meta-metamodel level of the standard OMG architecture. Note that there are no differences resulting from these two options with respect to the technical details of the OCL metamodel. In both cases the OCL metamodel is defined in terms of the MOF meta-metamodel.

Since OCL cannot be used without a concrete modeling language, we have chosen the first option and placed the OCL metamodel at the same level as the UML metamodel in order to achieve a tight integration. Figure 6.1

shows the relationship and the integration between both metamodels. OCL values are part of the M0 level contributing to the description of user objects and user data. Expressions and types are found on the M1 level as part of a model. The OCL metamodel on the M2 level therefore closes a gap in the metamodeling architecture and adds to the UML metamodel.

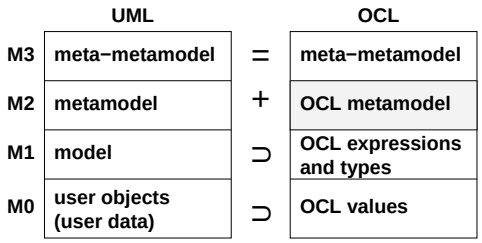


Figure 6.1: UML and OCL metamodel

The presentation of the OCL metamodel is structured in the following way. The metamodel is organized as a set of three UML packages containing a total of 54 classes. Each package describes a different aspect of the language, namely types, expressions, and values. For each package, we first present a class diagram defining the abstract syntax of a concept. Then we give an informal explanation of the diagram contents. Concrete examples are used to illustrate the application of the metamodel. Finally, a set of well-formedness rules specifies additional constraints on the metamodel. These rules are not complete, but they should give an idea of how the informal OCL rules given in [OMG99b] can be made precise by translating and applying them to the metamodel.

6.2 Structure of the Metamodel

The OCL metamodel is defined in a package called *OCL*. Figure 6.2 shows how this new package is related to the existing UML packages *Core*, *Data Types*, and *Common Behavior* [OMG99e, p. 2-7].

There are several dependencies among these packages. The *Core* package depends on *Data Types* where a general notion of expressions is defined. These expressions are used for modeling the concept of constraints in the *Core*. The *Data Types* package depends on OCL as soon as we use the Object Constraint Language for defining expressions. On the other hand, OCL depends on model elements (classes, attributes, etc.) of the *Core* package for building expressions. Finally, evaluating OCL expressions results in values. Both concepts are defined in the *Common Behavior* package, hence we have another dependency from *OCL* to *Common Behavior*.

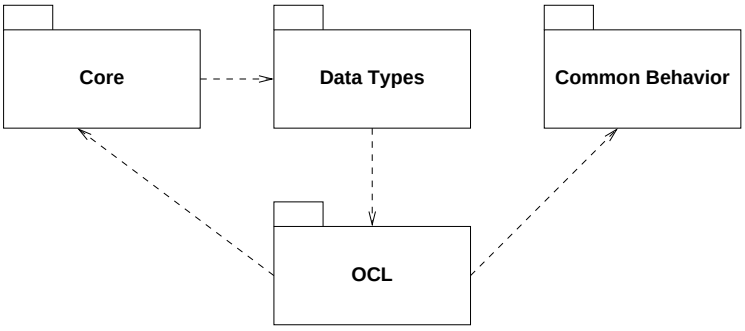


Figure 6.2: Dependencies among UML packages and OCL

There are more UML packages which use expressions and which are not included in Figure 6.2. For example, statecharts may contain boolean expressions for the specification of guard conditions. Of course, these packages will also indirectly (via *Data Types*) depend on and benefit from the *OCL* package when they use OCL for expressions.

The *OCL* package is further refined into three separate packages which model different aspects of OCL. Figure 6.3 shows the packages *Expressions*, *Types*, and *Values* as part of the OCL metamodel.

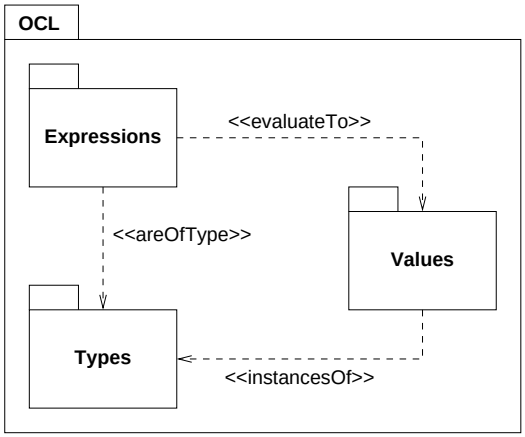


Figure 6.3: Package structure of the OCL metamodel

We have used special stereotypes for the dependencies to emphasize their different roles. In OCL, every expression has a well-defined type. Instances of types are values which result from the evaluation of expressions. Each of these three packages will be discussed in its own section in the following. Before doing so, we take a closer look at the relationship between the UML core and OCL constraints.

ProjectHeader: work 0.215 Mon, 20 Aug 2001 16:30:46 +0200 mr

6.3 Constraints

Constraints can be used to specify restrictions on UML model elements. The concept of constraints is defined in the *Core* package of the UML metamodel. A constraint may be attached to all kinds of model elements. For example, a class invariant can be defined by attaching a constraint to the class. The constraint specifies a boolean expression that has to be true for all instances of the class. UML does not prescribe the language or formalism for the boolean expression. In general, a constraint can be specified in natural language, OCL, or some other language.

The class diagram in Figure 6.4 provides the link between the UML metamodel and the OCL metamodel. It shows partial views of the *Core* package and the *Data Types* package [OMG99e, p. 2-13 and p. 2-75]. Only classes and relationships that are important for our purposes are displayed. Constraints[1] are ModelElements that specify restrictions on other model elements. A BooleanExpression forms the body of a Constraint. BooleanExpressions are just a special kind of Expression having a boolean result type. An expression can be defined by using OCL. In the UML *Data Types* package, Expression has attributes *language* and *body* where the body attribute keeps the textual representation of the constraint. We replace the *body* attribute by an optional component relationship to OclExpression. It is now possible to use OCL for defining expressions, but we are not forced to do so. Other languages could be integrated similarly.

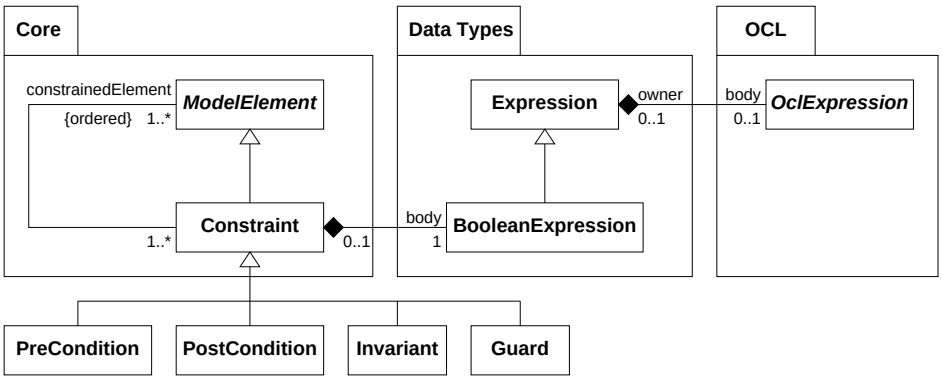


Figure 6.4: Integration of OCL expressions with standard UML packages

The abstract class OclExpression defines the set of all legal expressions in OCL. It is the top-level element of the OCL metamodel and will be further refined in Section 6.5. All possible kinds of expressions are specializations of OclExpression.

[1] Classes of the metamodel are set in a serif-less font in the following.

Constraints can appear in different contexts. They may be used to specify pre- and postconditions, invariants, and guards. Therefore, in Figure 6.4, we have specialized the Constraints class into corresponding subclasses PreCondition, PostCondition, Invariant, and Guard. The distinction is necessary because some OCL constructs are allowed only in certain contexts, for example, the `@pre` modifier makes sense only in postconditions. Alternatively, instead of modeling the different kinds of constraints as subclasses, we could also model this distinction as an enumeration in the Constraint class itself. However, our goal was to avoid changes to the original UML metamodel where possible.

The following well-formedness rules refer to some classes that are introduced in later sections.

Well-formedness rules:

[1] When a BooleanExpression is used in a Constraint and it is defined by an OclExpression, the Type of the OclExpression must be an instance of BooleanType.

```
context Constraint inv:
  let b = self.body.body in
    b.isDefined()
      implies b.resultType.oclIsKindOf(BooleanType)
```

[2] A PostCondition may only be attached to an Operation (which is defined in the UML *Core* package as a subclass of ModelElement).

```
context PostCondition inv:
  self.constrainedElement->forAll(
    me : ModelElement | me.oclIsKindOf(Operation))
```

6.4 Types

OCL is a typed language. Each expression has a type which is either explicitly declared or can be statically derived. Evaluation of an expression yields a value of this type. Therefore, before we can define expressions, we have to provide a model for the concept of types. A metamodel for OCL types is shown in Figure 6.5. Note that instances of the classes in the metamodel are the types proper (such as *Integer*) not instances of the domain they represent.

All available types in OCL are modeled as specializations of the abstract class Type. Concrete types are classified in the metamodel as follows.

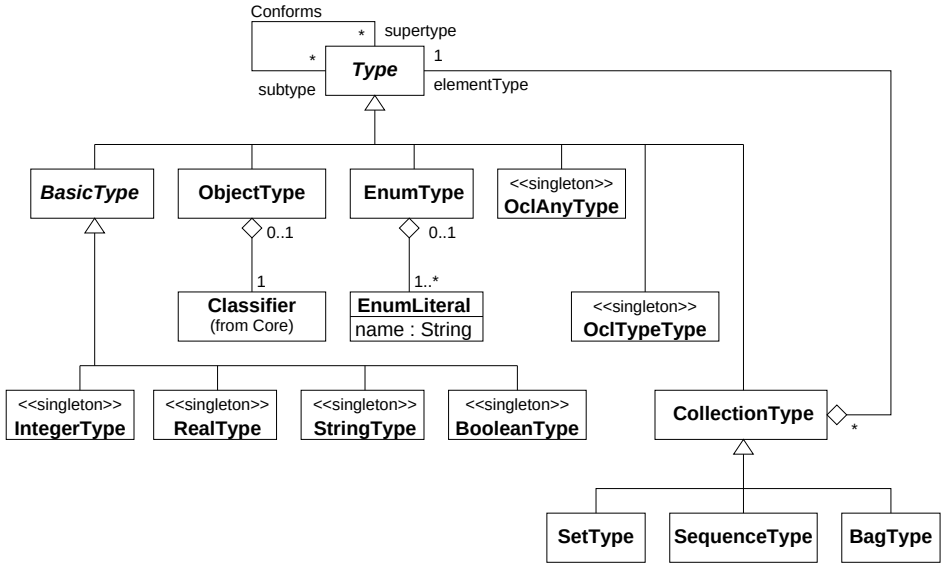


Figure 6.5: OCL Types package

- basic types: IntegerType, RealType, StringType, and BooleanType
- object types: ObjectType
- enumeration types: EnumType
- special types: OclAnyType and OclTypeType[2]
- collection types: CollectionType, SetType, SequenceType, and BagType

The first group models the basic types *Integer*, *Real*, *String*, and *Boolean*. We have marked the corresponding elements in the metamodel with a stereotype ≪singleton≫ to indicate that there is exactly one instance for each of these classes. For example, the one and only instance of IntegerType is the type *Integer*.

ObjectTypes are used to refer to Classifiers defined by users in a UML class model such as *Person* or *Company*. The meaning of special types is explained in [OMG99b]. EnumTypes are defined by a list of distinct literals like *Color* : {*red, green, blue*}. CollectionTypes are parameterized by an element type, for example, *Set*(*Integer*). Readers familiar with design patterns [GHJV95] will recognize the application of the composite pattern for modeling collection types. In our metamodel, there is no limit on the depth

[2]The strange looking name is a result of our naming scheme: For each OCL type we create a corresponding metamodel element by attaching the word Type to its name.

of nesting of collection types. Recall that OCL tries to avoid complex collections by a process called "automatic flattening". However, complex values may nevertheless result from the evaluation of some expressions (for example, by navigating more than one association with multiplicities greater than one). Thus, *before* "flattening" a complex value, we need to be able to specify its precise and complete (possibly nested) type.

The specialization of Type in the metamodel is used for classifying types by common properties. It is, however, not used for modeling subtype relationships which are defined by OCL type conformance rules. Rather, we use an association Conforms to model the subtype relation on types. This also allows us to express the type conformance rules for parameterized collection types (see the set of well-formedness rules below). As an example, consider the subtype relationship between the *Integer* and *Real* type: *Integer* is a subtype of *Real* such that an *Integer* value can be used anywhere where a *Real* value is expected. In the metamodel both types are generalized into a BasicType. The following constraint utilizes the Conforms association to enforce the requirement that *Real* is a supertype of *Integer*. It ensures that the set of supertypes of *Integer* includes the type *Real*.

```
context IntegerType inv:
  self.supertype->exists(t | t.oclIsTypeOf(RealType))
```

Example

We apply the metamodel to the OCL type $Set(Person)$. Figure 6.6 shows the representation of this type as a UML object diagram. The diagram is an instantiation of the *Types* metamodel. *Person* is a class defined in a user model. The class induces a corresponding object type that can be used in OCL. This object type is used to parameterize the set type.

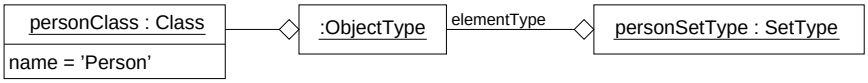


Figure 6.6: Object diagram for the collection type $Set(Person)$

In Figure 6.7, we continue the example and add another type $Set(Employee)$. In this example, class *Person* is a generalization of class *Employee*. The metaclass Generalization is defined in the UML *Core* package. We have also included in this diagram links of the Conforms association indicating type conformance. Since *Person* is a generalization of *Employee*, the object types have a similar relationship in OCL. Following from the type conformance

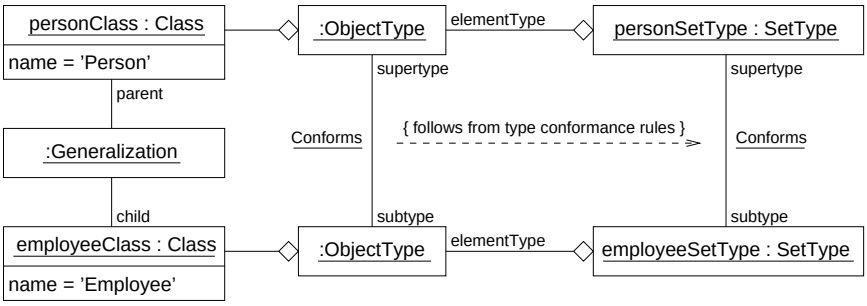


Figure 6.7: Object diagram illustrating type conformance rules between the collection types $Set(Person)$ and $Set(Employee)$

rules for collections (see [OMG99b, p. 7-20] and items 5 and 6 of the well-formedness rules below), $Set(Person)$ is also a supertype of $Set(Employee)$.

Most of the following well-formedness rules can be found as textual descriptions in the OCL documents [OMG99b, WK98]. With the metamodel, we are now able to formalize these rules as OCL constraints.

Well-formedness rules:

[1] OclAnyType is the supertype of all Types except for the collection types.

```
context Type inv:
  not self.oclIsKindOf(CollectionType)
    implies self.supertype->exists(t : Type |
      t.oclIsKindOf(OclAnyType))
```

[2] IntegerType conforms to RealType (OCL constraint given on page 122).

[3] All EnumerationLiterals of an EnumType are distinct.

```
context EnumType inv:
  self.enumLiteral->isUnique(el : EnumLiteral |
    el.name)
```

[4] Type conformance (represented by the association Conforms) is transitive.

```
context Type inv:
  Type.allInstances->forAll(t1, t3 : Type |
    Type.allInstances->exists(t2 : Type |
      (t1.subtype->includes(t2)
       and t2.subtype->includes(t3))
      implies t1.subtype->includes(t3)))
```

[5] A SetType *Set(T1)* conforms to a type *Set(T2)* if its element type *T1* conforms to *T2*.

```
context SetType inv:
  SetType.allInstances->forAll(s1, s2 : SetType |
    s1.elementType.supertype->includes(s2.elementType)
      implies s1.supertype->includes(s2))
```

[6] An ObjectType *O1* conforms to a type *O2* if their associated Classifiers *C1* and *C2* have a generalization relationship, that is, *C2* is a supertype of *C1*.

6.5 Expressions

In this section, we define the metamodel for OCL expressions. Perhaps surprisingly, it is quite difficult to say with reference to the OCL documentation what actually constitutes an expression. In [OMG99b], the term is used informally in a number of places and different contexts. We therefore follow our definition of the abstract OCL syntax in Chapter 5 where an OCL expression is defined to be one of the following.

1. A variable
2. A `let` expression
3. The application of an operation. We distinguish between
   - predefined operations
   - attribute operations
   - operations defined by a classifier
   - navigation by role names
   - constants
4. An `if` expression
5. A type test/cast expression
6. An `iterate`-based expression. We will refer to these expressions also as *query* expressions in the sequel.

The metamodel for expressions is shown in Figures 6.8, 6.9, 6.10 and 6.11. Due to the size of the class model we have decomposed it into four diagrams. The abstract class OclExpression is specialized into six fundamental expression classes according to the list given above. The first part in Figure 6.8

shows the basic structure of expressions and specializations for the six different expression kinds. The second part in Figure 6.9 shows elements for let, if, and type test/cast expressions. The third part in Figure 6.10 shows specializations of query expressions. Finally, the fourth part in Figure 6.11 shows specializations of operation expressions. In the following, we discuss each of the four class diagrams in more detail.

Core Elements

The first part of the metamodel shown in Figure 6.8 contains the core elements necessary to describe OCL expressions. All OclExpressions have a well-defined result Type. The class Type was defined in the *Types* package in Figure 6.5.

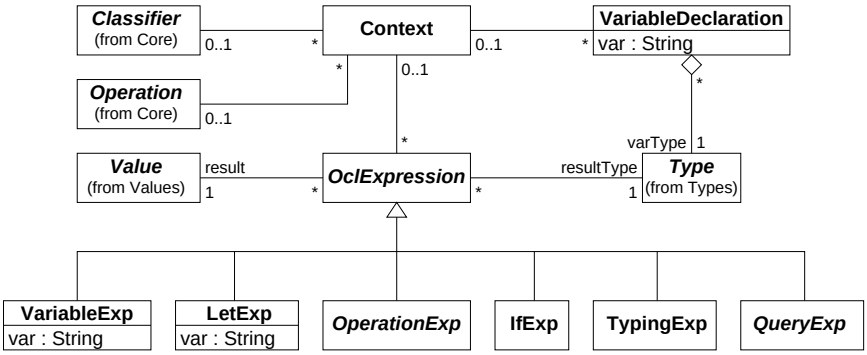


Figure 6.8: Metamodel for expressions (Part I)

Every OCL expression needs a context for evaluation. The context is given by the name of a classifier type such as *Person*. All occurrences of `self` variables are then bound to an instance of this type during evaluation. Furthermore, variables can be explicitly declared in a context, for example, `p : Person`. A context for a pre- or postcondition includes the signature of the operation to be constrained. Parameters of this operation introduce variable bindings that can be used just as any other variable in an expression. Variable declarations are represented by instances of VariableDeclaration. Each declaration associates a variable name with a Type. In the metamodel, the class Context connects each expression with a set of VariableDeclarations.

Let, If, Type test/cast

Figure 6.9 shows part of the metamodel for let, if, and type test/cast expressions. A LetExp expression declares and initializes a variable for use

in another expression. Details of the class VariableInitialization are shown in Figure 6.10. An IfExp expression has a *condition* and a *then* and *else* part. A TypingExp expression such as (e isTypeOf t) is applied to a source expression e and has an argument t which maps to a Type specifier.

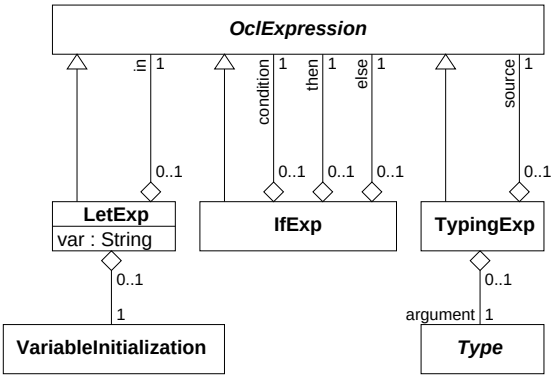


Figure 6.9: Metamodel for expressions (Part II)

Queries

The metamodel for query expressions is shown in Figure 6.10. The predefined query expressions `iterate`, `select`, `reject`, `collect`, `exists`, `forAll`, `isUnique`, and `sortedBy` differ from ordinary operations because they may contain language constructs for declaring and – in case of `iterate` – initializing variables that have special meaning in context of their argument expression.

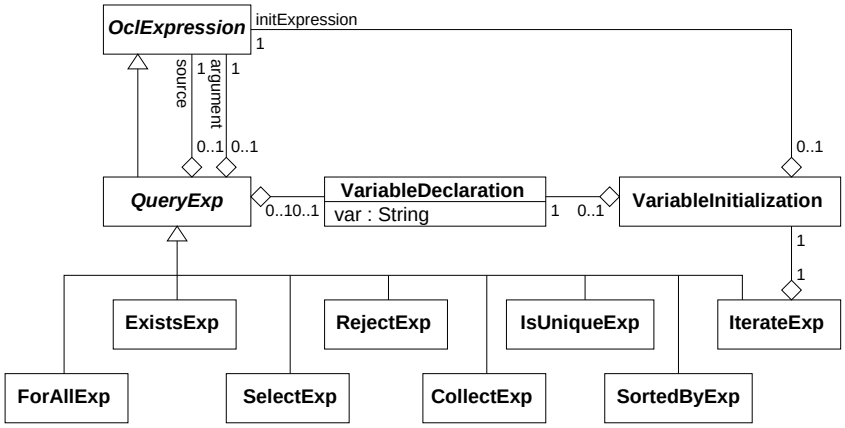
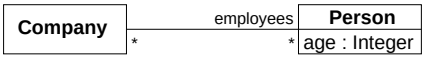


Figure 6.10: Metamodel for expressions (Part III)

The abstract metamodel class QueryExp represents common properties of query expressions. They all have a source expression resulting in a collection which forms the input of the query, and they have an argument expression which is evaluated for each of the source collection's elements. Furthermore, they optionally may have a VariableDeclaration introducing an identifier which may be referred to as a variable expression (VariableExp) in the argument expression.

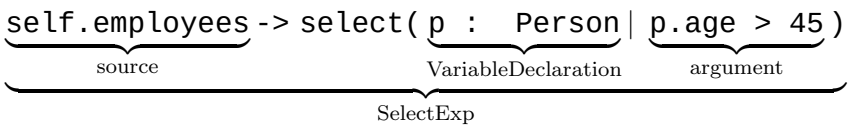
The more general `iterate` construct is a query expression (an IterateExp in the metamodel) which additionally has a mandatory variable initializer. A VariableInitialization is split into a declaration part and an initializing expression. We use the following simple class diagram to illustrate how an OCL expression maps to the metamodel.



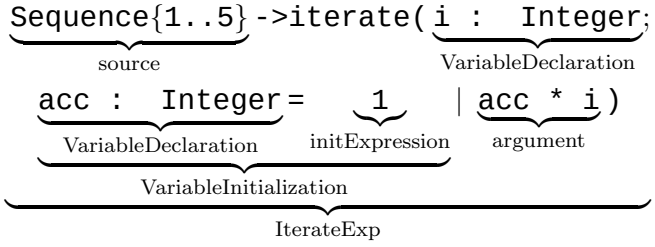
The following expression selects from the set of all persons working for a given company those who are older than 45.

```
context Company
  self.employees->select(p : Person | p.age > 45)
```

The query uses a select expression where a variable `p` of type *Person* is declared as part of the expression. This variable will be bound implicitly to each element of the source collection (`self.employees`) as the argument expression (`p.age > 45`) is evaluated for each of the collection's elements. In the metamodel, the expression can be represented as a SelectExp. We repeat the expression below and mark different components of the expression with labels referring to the class and role names used in Figure 6.10.



The following example illustrates the general structure of `iterate` expressions by means of an example.



Operations

A large number of expressions can be classified as applications of operations. The metamodel element OperationExp is an abstract class representing all kinds of operations in OCL. A refinement of OperationExp is shown in Figure 6.11.

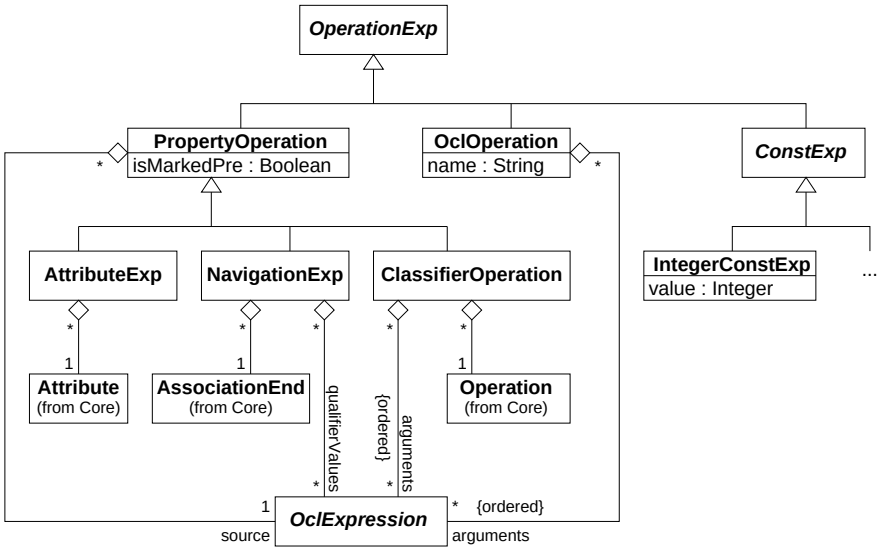


Figure 6.11: Metamodel for expressions (Part IV)

We first specialize OperationExp into PropertyOperations which are operations referring to state dependent properties of objects. We use the term property as it is introduced in [OMG99b] for attributes, association ends, and operations without side effects. All PropertyOperations have at least one argument: an OclExpression determining the source object. Furthermore, it is possible in postconditions to refer to a previous value of a property by postfixing it with the `@pre` keyword. The boolean attribute *isMarkedPre* indicates the reference to a previous value in a PropertyOperation.

An AttributeExp like `self.age` is a function with one argument: an OclExpression resulting in an object which owns the attribute. For the expression `self.age` the owner of the age attribute is given by an instance of VariableExp. The name of the operation is the name of the Attribute itself.

A NavigationExp utilizes associations originating from a classifier. An associated classifier is specified by its role name. The role name is part of an AssociationEnd in the *Core* package. When using qualified attributes on an AssociationEnd, a NavigationExp may also specify qualifierValues.

ClassifierOperations refer to Operations which are defined as classifier features in a user model, for example:

```
Employee::getSalary(d : Date) : Real
```

These kinds of operations may be used in OCL expressions, if they do not have any side effects that is the *isQuery* attribute of BehavioralFeature in the *Core* package is true [OMG99e, p. 2-25]. ClassifierOperations also allow a list of argument expressions.

Predefined OCL operations are modeled by the element OclOperation. These are characterized by an operation name and an argument list. Examples for predefined operations are: `+`, `-`, `*`, `<`, `>`, `size`, `max`, etc.

Finally, a further group of operations are those that produce constant values. We have modeled these separately for each of the basic types. For example, an IntegerConstExp simply contains an attribute *value* specifying the value of the constant to be created. The ellipsis in the diagram indicates the existence of further classes for the basic types *Real*, *String*, and *Boolean*. Constant collection values like `Set{1,2}` can be modeled as OclOperations. For example, a function mkSet($e_1 : T, \ldots, e_n : T) : \text{Set}(T)$ creates a set with n elements of type T. These functions are not visible to the user since they are only used for making the meaning of the literal notation explicit in the metamodel.

Example

We apply the metamodel for OCL expressions to the following expression that we already used above to illustrate the general approach.

```
context Company
  self.employees->select(p : Person | p.age > 45)
```

We can now give a precise representation of this expression in terms of an instance of the metamodel. Figure 6.12 shows an object diagram for the abstract syntax of the above expression as an instantiation of the metamodel for OCL expressions and types.

The object diagram basically shows an abstract syntax tree. The root of this tree is the `select` expression. It has three child branches. First, the source of the `select` operation is a collection resulting from the navigation expression `self.employees`. The second branch models the variable declaration `p : Person`. Finally, the third branch represents the expression `p.age > 45`. We do not show result types of expressions and the context information (given by class *Company*) in Figure 6.12 in order to keep the diagram readable. However, adding this information would be straightforward.

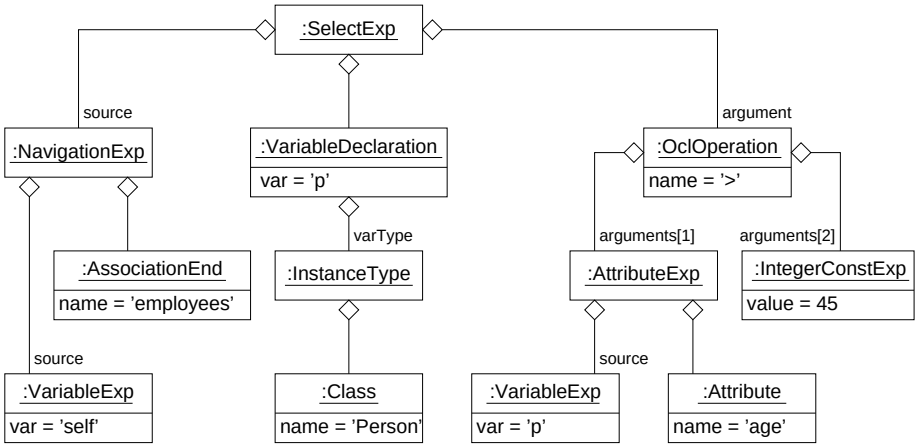


Figure 6.12: Example instantiation of the expressions metamodel

Well-formedness rules:

[1] A modifier `@pre` is only allowed in a PostCondition.

[2] The variable name used in a VariableExp must be part of a VariableDeclaration in an enclosing expression (or a parameter if the expression is attached to an operation as a pre- or postcondition). With other words, a variable must be declared before it may be used in an expression.

[3] The source expression of a QueryExp must have a collection type.

```
context QueryExp inv:
  self.source.resultType.oclIsKindOf(CollectionType)
```

6.6 Values

In the previous sections, we have defined a complete metamodel for the abstract syntax of OCL. Another important aspect is the interpretation of OCL expressions. The result of evaluating an expression is a value. Since our focus is on metamodeling issues, we do not consider the semantics of expressions here. A complete semantics as given in Chapters 4 and 5 provides a mapping from expressions to values. It is unclear, how this mapping can be reasonably represented by a metamodel. In the following, we will concentrate on the structure of values. A metamodel for values suitable for representing results from evaluating OCL expressions is shown in Figure 6.13.

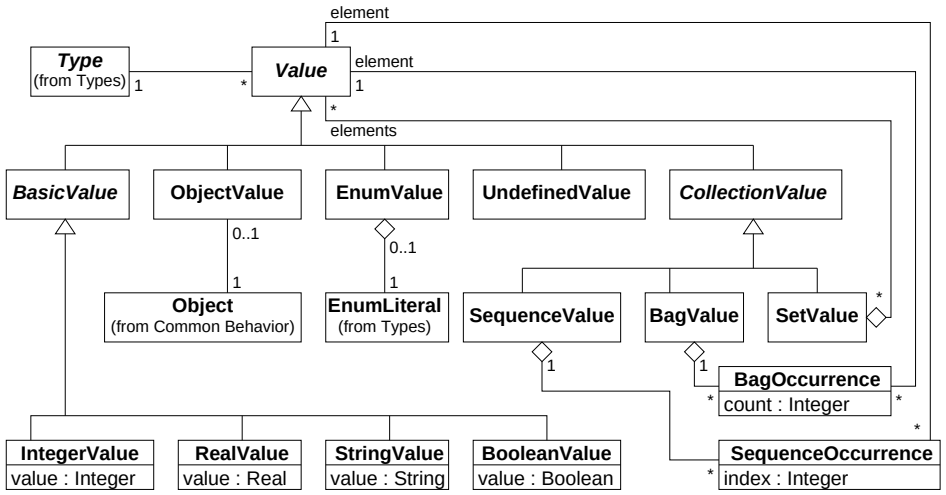


Figure 6.13: Metamodel for values

The general structure of values is very similar to the metamodel for types in Figure 6.5, since each value class represents the domain of a corresponding type. Value is the abstract base class of all kinds of values. Each value has a Type defining its properties. Values of basic types are IntegerValue, RealValue, StringValue, and BooleanValue. An ObjectValue is a reference to an Object which is defined as part of the UML *Common Behavior* package. An EnumValue is exactly one of the EnumLiterals used to define the corresponding type. Since the result of an OCL expression may be undefined, for example, as a result of a division by zero, a special UndefinedValue for each type is required.

We also need structures for the collection types *Set*, *Sequence*, and *Bag*. Values of these types may include other values as elements. For SetValue we can use an ordinary aggregation, SequenceValues and BagValues require additional classes SequenceOccurrence and BagOccurrence for dealing with multiple occurrences of equal values.

Example

We apply the metamodel to the value resulting from the OCL literal expression `Set{1,2}`. Figure 6.14 shows the representation of this set value containing two integer values. Each value has a link to its defining type on the left side. In this example, the hierarchical structure of values mirrors the hierarchical structure of composite types.

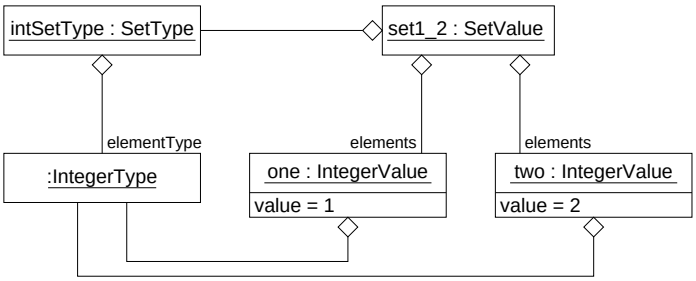


Figure 6.14: Object diagram for the value `Set{1,2}`

Summary

In this chapter, we presented a metamodel for the Object Constraint Language OCL. We demonstrated the application of the metamodel to concrete OCL expressions. The metamodel is defined as a set of class diagrams together with well-formedness rules. This style of presentation closely follows the style of the UML semantics document. The metamodel could therefore be easily integrated with the existing UML specification providing a uniform style of presentation for all parts of the Unified Modeling Language.

The metamodel delivers a more precise and detailed view of OCL. It contributes an abstract syntax for OCL that is not yet available in the UML standard. As a consequence from using only well-known modeling concepts of UML which are compliant with the Meta Object Facility (MOF), the metamodel can easily be read by everybody familiar with UML. The decomposition into packages for types, expressions, and values emphasizes the basic structure of OCL. The metamodel provides a blueprint for implementing OCL tools and contributes to a standardized metamodel-based exchange mechanism of OCL constraints. Some issues that remained unclear from the original documentation had to be resolved and solutions for these issues have been integrated into the metamodel. For example, we presented a concrete approach for connecting OCL with the UML core concepts. Furthermore, we gave an interpretation for several important OCL concepts such as the context of an expression, undefined values, and result values of expressions.

Chapter 7

Validating Models and Constraints

The Unified Modeling Language is a widely accepted standard for modeling software systems. A great number of CASE tools exists which facilitate drawing and documentation of UML diagrams. Many of the tools also offer automatic code generation and reverse engineering of existing software systems. However, often there is only little support for validating models during the design stage. Also, there is generally no substantial support for constraints written in the Object Constraint Language. While it seems feasible to translate constraints into program code as part of the code generation process, we argue that a model and its constraints should be validated before coding starts. Mistakes in the design can thus be detected very early, and they can easily be corrected in time.

We present an approach for the validation of UML models and OCL constraints that is based on animation. We developed a tool called USE (UML-based Specification Environment) for supporting developers in this process. The tool is available as Open Source [Ric01]. The main components of this tool are an animator for simulating UML models and an OCL interpreter for constraint checking. A UML model is taken as a description of a system. System states are snapshots of a running system. Snapshots can be manipulated, inspected, and checked for conformance with the model. The tool implements ideas and results from our work on formalizing the OCL in Chapters 3, 4, 5, and the metamodel for OCL in Chapter 6. A previous version of the work presented here has been published in [RG00c] and [RG00d].

Our validation tool can be generally applied to models from any domain. The metamodeling approach employed in UML allows us to treat metamodels in the same way as user models. As a special case, we have applied the tool to parts of the UML 1.3 metamodel and its well-formedness rules.

This is the first time (at least to our knowledge) that a tool enabled a thorough and systematic check of the OCL well-formedness rules in the UML standard. This also opens up a number of other useful applications. For example, the well-formedness of arbitrary models with respect to the UML standard can be automatically checked by validating them as instances of the UML metamodel.

There are currently only a few tools available which are specifically designed for analyzing UML models and OCL constraints. Probably, this is mostly due to the lack of a precise semantics of UML and OCL. A well-defined semantics is a prerequisite for building tools offering sophisticated analysis features. The following summarizes some work related to tools for checking UML designs. Alcoa is a tool for analyzing object models [JSS00]. It does not use OCL but has its own input language, Alloy, which is based on the specification language Z. RTUML [Mut00] focuses on real-time modeling aspects and offers a methodology for mechanized verification of design properties with PVS. An OCL toolset with a compiler and code generator combined with an OCL runtime library implemented in Java has recently been developed at the Dresden University [Fin00, HDF00]. A beta release of a commercial tool offering animation of UML models and OCL support is ModelRun from BoldSoft [Bol00].

The usability of tools supporting OCL greatly depends on efficient execution of OCL assertions. Hints for efficient execution of OCL are given in [CR99]. Problems related to automating animations of UML/OCL models are discussed in [OK99, Oli99]. A major problem – due to the inherent non-determinism of abstract specifications – is the complexity involved in generating a sequence of snapshots without user intervention. A different approach to validating UML models is described in [DdB00]. UML models are translated into the formal specification languages Z and Lustre. Then, tools already available for these languages are applied. In this approach, constraints on a class diagram are not specified with OCL but with Z. An animation system based on visual modeling languages is described in [BEE00]. The specification of dynamic behavior and animation of systems directly extends a visual modeling language based on formal graph transformation and graphical constraint solving techniques and tools.

The remainder of this chapter is structured as follows. In Section 7.1, we describe our general approach to validating UML and OCL. Section 7.2 gives an overview of the USE architecture. A case study is used in Section 7.3 to demonstrate the key features of our tool with respect to the validation process. Validation of dynamic aspects is described in Section 7.4. In Section 7.5, we report on applying the USE tool to the UML metamodel. This approach is generalized in Section 7.6 offering a number of interesting ways

for analyzing user designs. We close with a summary and draw some conclusions for future work.

7.1 The USE Approach to Validation

The goal of model validation is to achieve a good design before implementation starts. There are many different approaches to validation: simulation, rapid prototyping, etc. In this context, we consider validation by generating snapshots as prototypical instances of a model and comparing them against the specified model. This approach requires very little effort from developers, since models can be directly used as input for validation. Moreover, snapshots provide immediate feedback. They can be visualized using the standard notation of UML object diagrams – a notation most developers and potential users of a validation system are already familiar with.

The result of validating a model can lead to several consequences with respect to the design. First, if there are reasonable snapshots that do not satisfy the constraints, this may indicate that the constraints are too strong or the model is not adequate in general. Therefore, the design must be revised, for example, by relaxing the constraints to include the desired snapshots. On the other hand, constraints may be too weak, therefore allowing undesirable system states. In this case, the constraints must be changed to be more restrictive. Still, one has to be careful about the fact that a situation in which undesirable snapshots are detected during validation and desired snapshots pass all constraints does not allow a general statement about the correctness of a specification in a formal sense. It only says that the model is correct with respect to the analyzed system states. However, an advantage of validation in contrast to a formal verification is the possibility to validate non-formal requirements, and that it can easily be applied by average modelers without training in formal methods.

The diagram in Figure 7.1 illustrates the basic use cases for validating a model with USE. First, a model specification can be checked by the validation system. The *check specification* use case includes a syntax, type and semantic check. The syntax check verifies a specification against the grammar of the specification language which is basically a superset of the OCL grammar defined in [OMG99b, WK98] extended with language constructs for defining the structure of a model. The type check makes sure that every OCL expression can be correctly typed. Finally, a semantic check verifies a number of context-sensitive conditions. Among these conditions are the well-formedness rules defined as part of the UML Semantics [OMG99e]. An example for such a well-formedness rule is the requirement that a generalization hierarchy must not contain cycles.

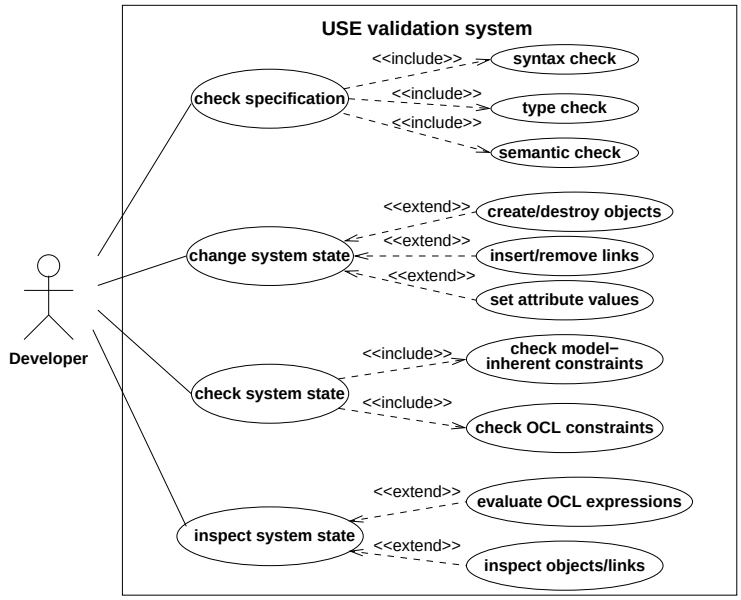


Figure 7.1: Use case diagram showing basic functionality of USE

When a specification has passed all checks, a developer may start to produce and *change system states*. A system state can be changed by issuing commands for creating and destroying objects, inserting and removing links between objects, and setting attribute values of objects. The developer can *check system states* at any time. A system state check includes two phases. First, all model-inherent constraints must be verified. A model-inherent constraint is a constraint which is inherent to the semantics of all UML models. For example, the set of links between objects is verified against the multiplicity specifications of the association ends. The number of objects participating in an association must conform to the multiplicities defined at the association ends. Second, if the developer has defined explicit OCL constraints, all the constraint expressions are evaluated. If any of the constraints is false or has an undefined result, the system state is considered ill-formed.

The *inspect system state* use case describes facilities for getting information about a system state. This is very important for helping a user to understand the effects of commands resulting in system state changes. Furthermore, when a constraint fails and a system state is found to be invalid, the developer has to find the reason for the failure. Inspecting a system state involves the inspection of individual objects, their attribute values and links. Another powerful way for inspection is the use of OCL as a query language. For example, consider a model where each object of class A must have at least one link to an object of class B, i.e., the association end at

class B has multiplicity `1..*`. If this multiplicity constraint is violated in some system state, the objects of class A which do *not* have a link to a B object can easily be found by the expression

```
A.allInstances->select(a | a.b->size = 0)
```

7.2 Architecture of USE

A high-level overview of the USE architecture is given in Figure 7.2. We distinguish between a *Description Layer* at the top, and an *Interaction Layer* below. The description layer is responsible for processing a model specification. The main component is a *Parser* for reading *Specifications* in USE syntax and generating an abstract syntax representation of a model. A USE specification defines the structural building blocks of a model like classes and associations. Furthermore, OCL expressions may be used to define constraints and operations.

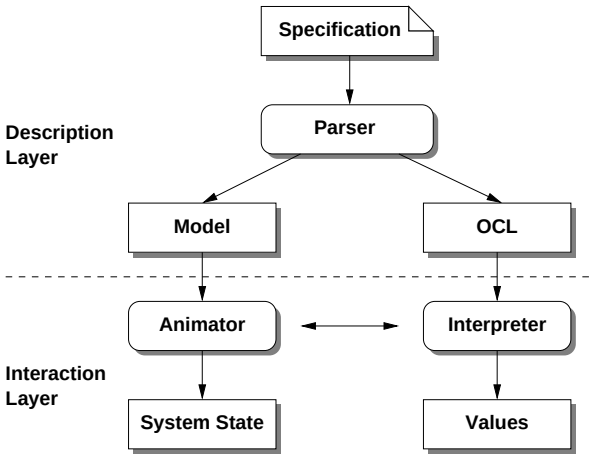


Figure 7.2: Overview of the USE architecture

The output of the parser is an abstract representation of a specification containing a *Model* and *OCL* expressions. The representation of the model is done with a subset of the Core package of the UML metamodel [OMG99e, p. 2-13]. The chosen subset corresponds to the Basic Modeling Language presented in Chapter 3 and excludes all model elements which are not required during the analysis and early design phase of the software development process. For example, model elements like Permission, Component, and Node seem to be more adequately applied in extended design and implementation models. The abstract representation of OCL expressions closely follows the OCL metamodel we have presented in Chapter 6 and [RG99a].

The *Interaction Layer* provides access to the dynamic behavior and static properties of a model. The main task of the *Animator* component is the instantiation and manipulation of *System States*. A system state is a snapshot of the specified system at a particular point in time. The system state contains a set of objects and a set of association links connecting objects. As a system evolves, a sequence of system states is produced. Each system state must be well-formed, that is, it must conform to the model's structural description, and it must fulfill all OCL constraints. Furthermore, a transition from one system state to the next must conform to the dynamic behavior specification given in form of pre- and postconditions.

The *Interpreter* component is responsible for evaluating OCL expressions. An expression may be part of a constraint restricting the set of possible system states. In order to validate a system state, the animator component delegates the task of evaluating all constraints to the interpreter. The interpreter is also used for querying a system state. A user may query a system state by issuing expressions that help inspecting the set of currently existing objects and their properties.

The *Model* and *OCL* branches in Figure 7.2 are tightly related to each other. For example, a model depends on OCL since operations of classes defined in a model may use OCL expressions in their bodies. A dependency in the other direction exists, because the context of OCL constraints is given by model elements. However, it is in general possible to define models which do not use OCL at all, and vice versa, there may be OCL expressions which just require an empty user model. For example, the realization of various general purpose algorithms with OCL like sorting or determining the transitive closure of a relation [MC99] is an interesting task on its own and can be done without the need for any particular model.

Animator and interpreter closely work together. The animator asks the interpreter for evaluating OCL expressions. On the other hand, the Interpreter needs information about the current system state, for example, when evaluating an expression which refers to the attribute value of an object.

7.3 Example Case Study

In this section, we demonstrate the validation of a UML model by means of a small case study. We start with a class diagram of a company model together with a few constraints.[1] The model is then specified in the textual USE notation. This specification serves as input to the validation tool. In an interactive session, a sequence of system states is produced by creating

[1] For a larger example, see Appendix B where a specification of the car rental case study introduced in Chapter 2 is given.

objects and links between them. Finally, we check a system state against the specification and show how the tool supports exploring the system state and helps in finding the reason for a constraint violation.

Figure 7.3 shows a UML class diagram of our example model. Employees have name, age, and salary attributes and work in departments. A department controls projects on which any number of employees can work. Both department and projects have attributes specifying their name and the available budget.

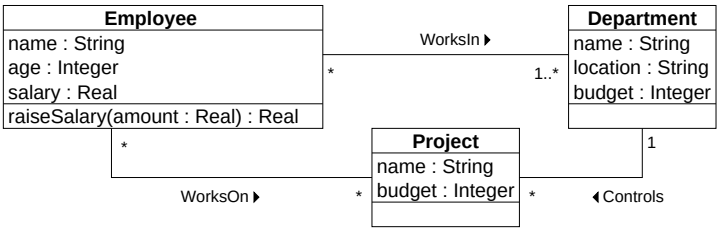


Figure 7.3: Class diagram of example model

The result of translating the class diagram into the textual USE notation is shown in Figure 7.4. The specification contains definitions for each of the classes and associations shown in the class diagram. The definition of a class includes a set of attributes and operation signatures. An association defines references to the participating classes for each association end. Multiplicity ranges are specified in brackets. Not used in this example, but also supported by the USE language, are UML features like generalization, different association types, role names, etc. The full grammar of USE specifications is given in Appendix A.

In order to make the example more interesting, we add some constraints which cannot be expressed graphically with the class diagram. The following five conditions have to be satisfied by a system implementing the given model.

[1] The salary and budget attributes are always positive.

[2] A department has at least as many employees as projects.

[3] An employee working on more projects than another employee gets a higher salary.

[4] The budget of a project must not exceed the budget of the controlling department.

[5] Employees working on a project must work in the controlling department.

```
model Company

class Employee
attributes
  name : String
  age : Integer
  salary : Real
operations
  raiseSalary
    (amount : Real) : Real
end

class Department
attributes
  name : String
  location : String
  budget : Integer
end

class Project
attributes
  name : String
  budget : Integer
end

association WorksIn between
  Employee[*]
  Department[1..*]
end

association WorksOn between
  Employee[*]
  Project[*]
end

association Controls between
  Department[1]
  Project[*]
end
```

Figure 7.4: USE specification of the example model

For each of these constraints we have specified OCL expressions that are used as invariants on the classes. We continue the specification begun in Figure 7.4 with a section defining the set of constraints shown in Figure 7.5. Each invariant is named for allowing an easy reference to the list above. Note that constraint [1] actually maps to three OCL invariants (`i1a`, `i1b`, `i1c`) since it states a condition on each of the three classes.

We can run the USE tool with the specification and start with an empty system state where no objects and no association links exist. In the next step, we are going to populate the system with objects and link them together. There are basically three kinds of commands which allow us to modify a system state: (1) creating and destroying objects, (2) changing attribute values, and (3) inserting and deleting association links. The tool provides two different kinds of user interfaces for these commands which can be used in parallel.

- A graphical interface supports an intuitive approach. For example, a new object can be created by simply selecting a class name and dragging it onto an object diagram.

```
constraints

context Department
  inv i1a: self.budget >= 0
  inv i2:  self.employee->size >= self.project->size

context Employee
  inv i1b: self.salary >= 0
  inv i3:  Employee.allInstances->forAll(e1, e2 |
             e1.project->size > e2.project->size
               implies e1.salary > e2.salary)

context Project
  inv i1c: self.budget >= 0
  inv i4:  self.budget <= self.department.budget
  inv i5:  self.department.employee->includesAll(self.employee)
```

Figure 7.5: USE specification of OCL constraints

- A scripting interface provides a shell-like environment for experts. It also allows the automation of many validation tasks, for example, scripts can be processed without requiring any user interaction. A grammar describing the syntax of commands is given in Appendix A.3.

Figure 7.6 shows a screenshot of USE visualizing a system state after several objects and links have been created. On the left side, the user interface provides a tree view of the classes, associations, and constraints in the model (parts referring to pre- and postconditions are explained in the next section). The pane below shows the definition of the currently selected component (the invariant `Project::i4`). The pane on the right provides a workspace and contains several different views of the current system state. These views are automatically updated as the system changes. A user can choose from a number of different available views each focusing on a special aspect of a system state. In this example, there are views showing an object diagram, a list of class invariants with their results, and an automatically generated sequence diagram displaying the message flow resulting from an operation call.

The highlighted constraint `Project::i4` in the class invariant view has the result false indicating that the current system state does not conform to the specification. The plain information that an invariant has been violated is usually not very helpful in finding the reason for the problem. The invariant from the example is quite short (see Figure 7.5). We can infer from the specification that there must be at least one project with a budget exceeding the budget of its controlling department. We could proceed by inspecting all

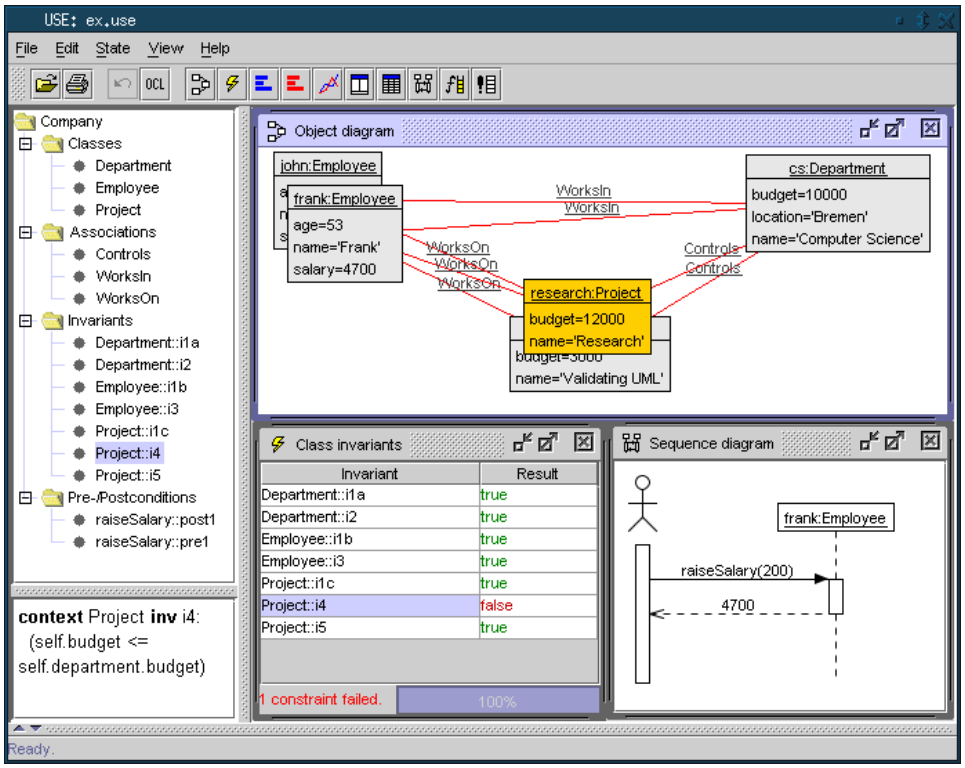


Figure 7.6: USE screenshot

projects until we find one violating the constraint. For larger systems, this quickly becomes a laborious task. Our tool therefore offers special support for analyzing OCL expressions. The details of evaluating an OCL expression can be examined by means of an evaluation browser which provides a tree view of the evaluation process. Figure 7.7 displays such a browser for the failing invariant.

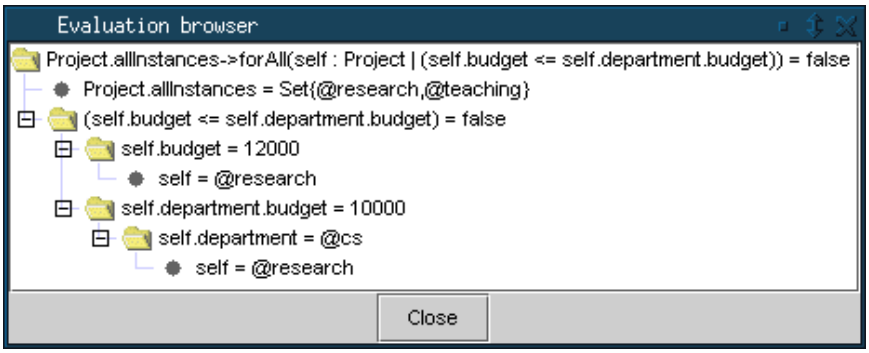


Figure 7.7: OCL evaluation browser

Each node in the browser shows an OCL expression and its result after evaluation. The root node shows the complete OCL expression defined as the body of the invariant. The original expression is first expanded into a self-contained expression which does not require a context. This transformation was explained on page 98 in Chapter 5. An invariant for a classifier C having the general form

```
context C inv:
  <expr-with-self>
```

can be transformed into an equivalent expression without a context specification like this:

```
C.allInstances->forall(self : C | <expr-with-self>)
```

Child nodes in the evaluation browser represent sub-expressions which are part of their parent node's expression. Evaluating the `forAll` expression requires the evaluation of the source collection (given by the sub-expression `Project.allInstances`) and the argument expression (`self.budget <= self.department.budget`). The evaluation of a `forAll` expression stops immediately when the condition is false for an element of the source collection. By looking at the current binding of `self`, we can conclude that it is the budget of the "research" project which invalidates the whole invariant, because it is greater than the budget of the controlling department.

USE also provides another simple way to "debug" constraints. For each invariant it can determine the set of individual instances violating an invariant. Similar to the transformation of invariants described above, the following expression is generated and evaluated for an invariant.

```
C.allInstances->reject(self : C | <expr-with-self>)
```

This expression results in the set of all C objects that do *not* satisfy the invariant. For the example, we get the following output:

```
checking invariant (6) 'Project::i4': FAILED.
  -> false : Boolean
Instances of Project violating the invariant:
  -> Set{@research} : Set(Project)
```

The most flexible option, however, is given by a facility for evaluating arbitrary OCL expressions (see Figure 7.8). The dialog for evaluating OCL expressions allows ad-hoc queries useful for navigating and exploring a system state at any time. It can also be used, for example, to interactively test new constraints before they are added to a model specification.

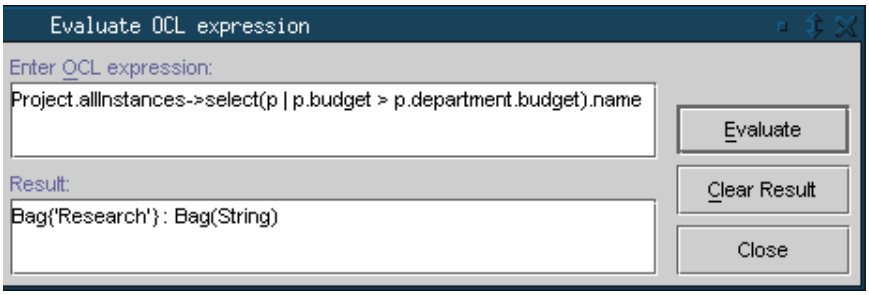


Figure 7.8: OCL evaluation dialog

7.4 Pre- and Postconditions

In this section, we focus on support for validating pre- and postconditions. This new feature was not available in the version of the tool presented in [RG00c] but has recently been added. The behavioral interface of objects is defined by operations in UML. Constraints on the behavior are specified in OCL by means of pre- and postconditions. Such a constraint defines a contract that an implementation of the operation has to fulfill [Mey97]. Pre- and postcondition specifications also generally provide the possibility for verifying the correctness of an operation implementation, for example, in the style of Hoare logic [Hoa69].

The USE tool allows the early validation of pre- and postconditions. An implementation of operations in a programming language is not necessary for this task. In order to demonstrate this, we slightly extend the model introduced in the previous section. We add a specification of pre- and postconditions to the raiseSalary operation defined in class *Employee*.

```
-- If the amount is positive, raise the salary by
-- the given amount and return the new value.
context Employee::raiseSalary(amount : Real) : Real
  pre:  amount > 0
  post: self.salary = self.salary@pre + amount
     and result = self.salary
```

The automatically generated sequence diagram in Figure 7.6 shows the message flow between objects. In this example, the operation raiseSalary with a parameter value 200 has been called for the employee Frank who previously had a salary of 4500. An interactive command window (not shown in the figure) reports the success of the pre- and the postcondition. In case of a failing precondition the operation could not have been entered at all. A failing postcondition would additionally be visualized with a red return arrow

in the sequence diagram. In both cases, a detailed report on the evaluation of expressions gives hints to the user why the conditions failed. The commands (suitable for the scripting interface) that simulate the operation call are the following.

```
-- call operation raiseSalary
!openter frank raiseSalary(200)
!set self.salary = self.salary + amount
!opexit 4700
```

An operation call is simulated by first issuing an `openter` command with a source expression, the name of the operation and an argument list. The `openter` command has the following effect.

1. The source expression is evaluated to determine the receiver object.
2. The argument expressions are evaluated.
3. The OCL variable `self` is bound to the receiver object and the argument values are bound to the formal parameters of the operation. These bindings determine the local scope of the operation.
4. All preconditions specified for the operation are evaluated.
5. If all preconditions are satisfied, the current system state and the operation call is saved on a call stack. Otherwise, the operation call is rejected.

The side effects of an operation are specified with the usual USE commands for changing a system state. In the example, the `set` command assigns a new value to the salary attribute of the employee. After generating all side effects of an operation, the operation can be exited and its postconditions can be checked. The command `opexit` simulates a return from the currently active operation. The result expression given with this command is only required for operations that specify a result value. The `opexit` command has the following effect.

1. The currently active operation is popped from the call stack.
2. If an optional result value is given, it is bound to the special OCL variable `result`.
3. All postconditions specified for the operation are evaluated in context of the current system state and the pre-state saved at operation entry time.

4. All variable bindings local to the operation are removed.

In our example, the postcondition is satisfied and the operation has been removed from the call stack. We give another example that shows how operation calls may be nested in the simulation. It also shows that postconditions may be specified on operations without side effects. An OCL expression is given to describe the computation of a side effect free operation. In the example, we use a recursive definition of the factorial function.

```
model NestedOperationCalls

class Rec
operations
  fac(n : Integer) : Integer =
    if n <= 1 then 1 else n * self.fac(n - 1) endif
end

constraints

context Rec::fac(n : Integer) : Integer
  pre:  n > 0
  post: result = n * self.fac(n - 1)
```

The postcondition of the operation `Rec::fac` reflects the inductive case of the definition of the factorial function. The following commands show the computation of 3!.

```
!create r : Rec
!openter r fac(3)
!openter r fac(2)
!openter r fac(1)
!opexit 1
!opexit 2
!opexit 6
```

The operation calls are exited in reverse order and provide result values that satisfy the postcondition. Figure 7.9 shows the sequence diagram generated from this call sequence. The stacked activation frames in the diagram emphasize the recursion.

7.5 Validating the UML Metamodel

The USE validation tool can be generally applied to models from any domain. As a special case, we have applied it to the complete UML 1.3 meta-

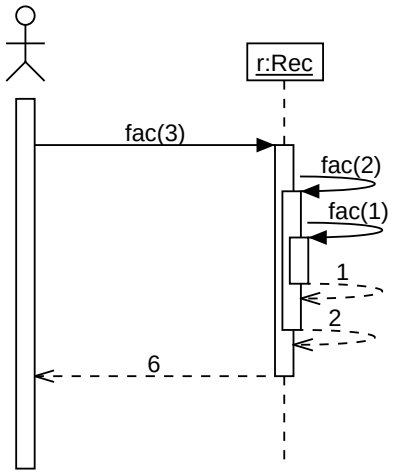


Figure 7.9: Sequence diagram for recursive operation call

model and its well-formedness rules.[2] The tool enabled a systematic check of the OCL well-formedness rules in the UML standard. This section describes the procedure for checking the metamodel and some results.

In the first step, we had to translate the class diagrams defining the UML metamodel [OMG99e] into the textual USE notation. Next, all well-formedness rules as well as additional operations specified in [OMG99e] were added. Some minor syntactical changes required by the USE syntax were necessary, for example, identifiers which are reserved keywords in USE had to be renamed.

The final specification[3] has 95 classes and 98 associations (see Table 7.1). There are 152 invariants defining well-formedness rules and 42 additional operations defining frequently required expressions as operations. The result is a total of 194 OCL expressions some of which are fairly complex.

Classes	Associations	Invariants	Operations	OCL Expressions
95	98	152	42	194

Table 7.1: Number of analyzed elements in the UML metamodel

The expressions in the UML document had a number of errors which could quickly be located by analyzing the error messages signaled by the USE parser. Some errors could easily be corrected, others indicated more serious problems with the constraints. We classified the problems into the following categories (with increasing severity) and give an example for each category.

[2] This extends the analysis in [RG00c] which only considered the Core package of UML.
[3] Available as part of the USE distribution [Ric01]

A class name together with a number in brackets refers to the corresponding well-formedness rule in [OMG99e].

E1: *Syntax errors*

- Example: wrong spelling of keywords and standard operation names
(`Association[3]`, `AssociationEnd[1]`)

E2: *Minor inconsistencies*

- Example: there is no operation `max` defined on Multiplicity
(`AssociationEnd[2]`)

E3: *Type checking errors*

- Example: union of sets with incompatible element types
(`Classifier[4]`, `Classifier[5]`)
- Example: implicit collect expression returns a bag not a set
(`ModelElement::supplier()`)

E4: *General problems*

- Example: the operation `contents()` in class Namespace has syntax errors and its description is identical to the description of the operation `allContents()`. It remains unclear how these operations should look like.

The results from analyzing the OCL well-formedness rules are summarized in Table 7.2. We found that there were errors in 50% of all expressions (97 out of 194). Some expressions had two or more errors belonging to different categories. Most errors are found in category E1. These can be fixed without much effort. The other errors generally require more work and detailed knowledge of the metamodel.

OCL Expressions	No Errors	Errors	Errors in Category			
			E1	E2	E3	E4
194	97	97	38	16	37	28

Table 7.2: Results from analyzing OCL expressions in the UML metamodel

It is not very surprising that a tool-based mechanical check of OCL expressions greatly helps in finding frequently occurring errors such as spelling mistakes. The fact that OCL provides strong typing also helps in getting complex expressions right. Another general observation that we have made

is related to the style of the OCL syntax. In some cases, a single notation is used for many different things. This makes it sometimes quite difficult to understand an expression and requires a lot of context knowledge. From a human's point of view this complicated the task of reading, understanding and checking OCL expressions. Consider, for example, the definition of the operation `allParents` in class GeneralizableElement:

```
allParents : Set(GeneralizableElement);
allParents = self.parent->union(self.parent.allParents)
```

The syntax of the expression `self.parent` is the same for referring to an attribute, an operation, or a role name of an associated class. Furthermore, `parent.allParents` may again be an attribute reference, an operation call, or a navigation by role name. Additionally, it may be an implicit collect expression written in shorthand notation. To find out which case is actually present, one has to look at the attributes, the operations and associations of all the referenced classes. However, this may still not be sufficient since all these features might be defined in superclasses so that the generalization hierarchy also has to be taken into account. We therefore "re-engineered" most expressions to an explicit form making the intended meaning much clearer. The example from above was augmented with an explicit collect expression and a flattening operation. Also, operation calls can now be spotted immediately due to the use of parenthesis. Actually, the expression in the UML standard is not type correct since the expression type is a bag. The final conversion of the result to a set is therefore required by the declaration.

```
allParents = self.parent()->union(
  self.parent()->collect(g |
    g.allParents())->flatten)->asSet
```

7.6 "Meta-Validation"

We found the USE tool to be very beneficial for understanding and analyzing the well-formedness rules of the UML metamodel. A number of errors in the OCL expressions could be quickly located and corrected. This might not only be useful for improving the quality of the UML standard, but also implies another very nice application: in principle, any UML model can be checked for conformance to the UML standard. Conformance in this context has the following meaning. A model conforms to the UML standard if it can be represented as an instance of the UML metamodel. The general idea is to (1) import a UML model (preferably in XMI representation [OMG99f]), (2) traverse the model and produce and execute a sequence

of USE commands for instantiating the model elements as objects of the UML metamodel, and (3) check all constraints on the resulting snapshot. All these steps can be done mechanically. If the last step fails, the model is not conform to the UML standard.

Figure 7.10 illustrates how this process of "meta-validation" can be done with USE. The general approach is very similar to the standard validation task. However, instead of validating user *data*, this time we are checking user *models*. Thus, we are moving up one level with respect to the metamodeling architecture employed in UML.

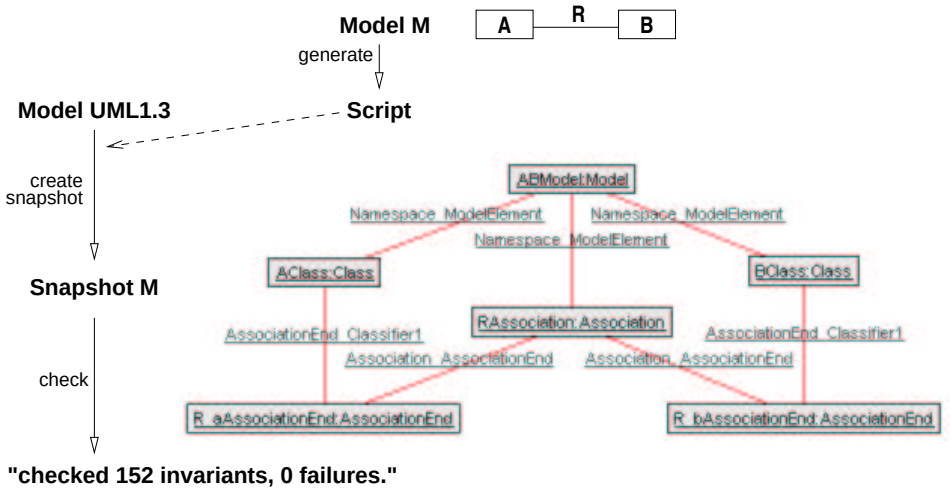


Figure 7.10: Sketch of the meta-validation process

As the figure demonstrates, we start with a user model M containing two classes A and B and an association R. The USE tool provides a feature that automatically derives and generates a script from this model. This script contains all commands necessary for creating an instance of the UML 1.3 metamodel that represents the model M. In the next step, the specification of the UML metamodel is loaded and the script is executed. The result is a snapshot representing M as an instance of the UML metamodel. The snapshot is shown by an object diagram in the figure. Objects in the snapshot are instances of metamodel classes. There are objects for the model M, the two classes A and B, the association R and two association ends connecting the classes with the association. Finally, we let the USE tool check all the 152 well-formedness rules defined for UML models. In this example, all of these invariants are satisfied. Provided that the previous transformation steps are correctly implemented, we have a very strong evidence that the user model M indeed is a well-formed UML model.

Our approach of "meta-validation" has some very practical benefits.

- It provides a flexible way for doing conformance tests for CASE tools. In order to test a specific tool, we can apply the procedure described above to any model M produced by the tool. For an input model M the output of the procedure is true if the model is UML conform, or false if the model is not conforming.
- It can be integrated with CASE tools enabling dynamic checks of well-formed rules at design time. Thus, a designer gets immediate feedback when a model is ill-formed.
- The well-formedness rules do not have to be hard-coded in tools. Since they are written in a specification language they can easily be changed and extended. This allows for faster adaptation to evolving standards.

Using the metamodel for analyzing user models is not only beneficial for verifying the well-formedness of a model. The fact that OCL is a general expression language implies that it can also be used to query models for various aspects. For example, the following expression finds all binary associations in a model.

```
Association.allInstances->select(a |
  a.connection->size = 2)->collect(a | a.name)
```

If we apply this query to the model M given in Figure 7.10, we get the result `Bag{'R'}` indicating that the association R is the only binary association in the model. Similarly, the following expression returns the number of reflexive associations by selecting only those associations whose ends have the same type.

```
Association.allInstances->select(a |
  a.connection->forAll(ae1, ae2 |
     ae1.type = ae2.type))->size
```

In general, arbitrarily complex queries are conceivable allowing a sophisticated analysis of design models. This method can be used, for example, to gather data that might serve as input for quantitative measures contributing to an object-oriented design metrics [CK94, Whi96, HS96]. The goal of such a design metrics is to improve the overall quality of a software design. Various metrics have been proposed that may help in refactoring a design. For example, the number of methods per class indicate the complexity of a class and the effort to expect for maintaining derived classes. Furthermore, a large number of children classes and a deep inheritance graph makes it more complex to predict the behavior of inherited methods.

Summary

In this chapter, we presented a tool-based approach to validating UML models and OCL constraints. The ideas presented here have been implemented in the USE tool. The tool has reportedly been applied in industry and research, and in teaching lightweight formal methods in software engineering. The functionality of USE was shown by means of use cases and an example case study. We applied the tool for checking the UML metamodel which makes extensive use of OCL constraints. As a result we could identify a number of errors in the standard document. Using the metamodel as a specification of arbitrary UML models, the tool enables a mechanical check for conformance of these models with the standard. The USE tool and our specification of the UML metamodel has also been used by others to define a metamodel for an Action Semantics to be submitted as response to a "Request for Proposal" by the OMG [OMG98].

The OCL parser and interpreter that is part of USE implements most of the core features of OCL like expression syntax, strong type checking, evaluation of expressions, and validation of pre- and postconditions. We have implemented almost all of the more than 100 standard operations on predefined OCL types. Some OCL features not yet implemented include some rarely used features like the syntax of path expressions, qualifiers, and type checking of empty collection literals.

There are several possible future extensions which would fit within the USE framework. For example, it would be nice if the animation could be automated to some extent by deriving test cases from the model. First steps towards test and validation automation have been pursued in [Boh01]. A language for generating snapshots was designed, implemented and integrated with USE.

Another extension could apply the validation techniques of USE to implementations of a model. Program code could be generated which mirrors the state of a system at runtime. The state traces can be observed and analyzed in parallel with USE. With this approach there is no need for transforming OCL expressions into program code since the interpretation of expressions is already part of USE. Also very useful would be an analysis of OCL constraints with respect to properties like consistency. However, there is currently no clear definition of what it means for a set of OCL constraints to be consistent. A discussion of this can be found in the OCL Semantics FAQ [CKW+00b].

Chapter 8

Conclusions

In this thesis, a precise approach to validating OCL constraints and UML models was introduced. The main results are briefly summarized in Section 8.1. In Section 8.2, we discuss our contributions and draw some conclusions. We close with an outline of future work.

8.1 Summary

We started with an overview and a discussion of the UML and OCL language definitions and found that the metamodeling approach to defining UML provides a more precise definition compared with most of its predecessor languages. Nevertheless, the UML definition is still a compromise between formality and the goal of understandability. The same observation holds for the Object Constraint Language OCL. While the UML makes at least a clear distinction between abstract syntax and notation, there is no such distinction being made for OCL. Only an informal description, examples and a definition of the concrete syntax is given in the standard document. Consequently, we could identify a number of issues resulting from underspecification and ambiguous explanations.

A precise foundation for OCL requires the consideration of essential parts of UML which provide the context for constraints. We therefore defined the notion of an object model. A formal definition of object models containing classes, attributes, operations, generalization hierarchies, and associations was given. These basic concepts constitute fundamental modeling elements relevant in the analysis and early design phase of a software development process. The syntax of each of these concepts was defined. A precise semantics was given by an interpretation of object models that maps elements of the syntax to a semantic domain. The concept of system states representing snapshots of a system was introduced. A snapshot contains objects,

links, and attribute values and can be generally visualized by UML object diagrams.

The formalism for object models provided the necessary framework for a detailed discussion of OCL. We started by defining the OCL type system including basic, enumeration, and object types. Complex types were introduced with collection types. We declared *OclAny* as a special type and defined a type hierarchy based on a subtype relation. We also proposed some possible extensions to the standard type system. Extensions included tuple types, association types, and user-defined data types. Based on the standard types, we finally defined the data signature $\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$. For a given object model $\mathcal{M}$, a data signature describes the set of types, the type hierarchy, and the set of operations over these types. The data signature provides the foundation for defining the language of OCL expressions. The formalization of OCL was completed by giving an abstract syntax and semantics of OCL expressions. We also defined the context of expressions and gave interpretations for common shorthand notations. The meaning of constraints such as invariants and pre- and postconditions was made precise.

Building upon the precise OCL definition, we presented a metamodel for the Object Constraint Language. We demonstrated how concrete OCL expressions are represented as instances of the metamodel. The metamodel was defined as a set of class diagrams together with well-formedness rules. This style of presentation closely follows the style of the UML semantics document. The metamodel could therefore be easily integrated with the existing UML specification providing a uniform style of presentation for all parts of the Unified Modeling Language including OCL.

The metamodel delivers a more precise and detailed view of OCL. As a consequence from using only well-known modeling concepts of UML which are compliant with the Meta Object Facility (MOF), the metamodel can easily be read by everybody familiar with UML. The decomposition into packages for types, expressions, and values emphasizes the basic structure of OCL. Some issues that remained unclear from the original documentation had to be resolved and solutions for these issues have been integrated into the metamodel. For example, we presented a concrete approach for connecting OCL with the UML core concepts. Furthermore, we gave interpretations for several important OCL concepts such as the context of an expression, undefined values, and result values of expressions.

In order to demonstrate the practical applicability of our work, we have realized substantial parts of it in the USE tool supporting the validation of models and constraints. Design specifications can be “executed” and animated thus providing early feedback in an iterative development process. The tool has reportedly been applied in industry and research, and in teaching lightweight formal methods in software engineering. The functionality

of USE was shown by means of use cases and an example case study. We applied the tool for checking the UML metamodel which makes extensive use of OCL constraints. As a result we could identify a number of errors in the standard document. Using the metamodel as a specification of arbitrary UML models, the tool enables a mechanical check for conformance of these models with the standard.

8.2 Conclusions and Future Work

We have made the following contributions to the rigorous development of software systems with UML and OCL.

- Parts of the UML core were formally defined allowing an unambiguous interpretation of essential concepts in software modeling.
- A precise definition of OCL avoiding ambiguities, under-specifications, and contradictions was given. Several lightweight extensions were proposed to improve the orthogonality of the language.
- The integration of UML and OCL was improved by making the relationships and dependencies between both languages explicit.
- A solid foundation for tools supporting analysis, simulation, transformation and validation of UML models with OCL constraints was developed. The feasibility of results presented in this work was shown with the realization of the USE tool.
- The USE tool has been used to validate the well-formedness rules in the UML standard. The results provide input for improving future UML versions.

The precise definition of OCL and core parts of UML enables an unambiguous interpretation of models and constraints. We have seen that both languages are closely related to each other. A definition of OCL cannot be considered without looking at parts of UML providing the context for constraints. Our definition avoids the problem of circularities arising in a pure metamodel approach. For example, even if we are using OCL to state well-formedness rules on the OCL metamodel, the semantics of these constraints is well-defined in terms of the set-theoretic formalization. Usage of OCL for defining metamodels like UML, MOF or CWM [OMG00a] therefore also improves the quality of other modeling languages.

An important topic for future work is related to the forthcoming UML 2.0 standard. Currently, submissions are being worked out by various groups

following a *Request For Proposal* (RFP) by the OMG. A major architectural change in the way how UML is defined is to be expected. Recent experience has shown that it is difficult to define a single language accommodating all the different needs of application areas such as real-time modeling or business modeling at the same time. Therefore, better mechanisms for adapting UML to different requirements are necessary. For example, profiles and prefaces have been suggested as a way to customize UML making it effectively a family of languages [CKM+99b, Coo00]. It has to be investigated how the architectural change of UML will affect our formal framework. However, because compatibility with previous versions is a major goal in the development of standards, we do not expect fundamental changes. Moreover, the many works on a precise UML definition during the last years now seem to have a positive influence on future UML versions as indicated, for example, by two quotes from the UML 2.0 Request For Proposal [OMG00b]:

> Proposals shall enforce a clear separation of concerns between the specification of the metamodel semantics and notation, including precise bi-directional mappings between them.

The design of a small kernel language is promoted by the following request:

> Proposals shall restructure the UML metamodel to separate kernel language constructs from the standard elements that depend on them.

A separate Request For Proposal has been issued for OCL [OMG00c]. This RFP references [RG99a] and explicitly solicits the submission of an OCL metamodel: "Proposals shall provide a metamodel for OCL that integrates with the UML metamodel." Our OCL metamodel meets these requirements, and it is currently under discussion in a work group preparing a submission for the OCL 2.0 RFP.

In the following, we discuss some general important issues that should be investigated in future work.

- A notion of consistency of OCL constraints needs to be found. A proposal related to the consistency of invariants has been made in [CKW+00b]: "For each class, there should be at least one possible system state in which the set of object instances of the class is nonempty." This requirement may also be applied to postconditions. Checks for contradictions between postconditions and invariants are very useful for maintaining the overall consistency of a class specification. Closely related is the question of how to translate between both kinds of constraints. For example, invariants that are specified for a whole class should, in general, also be expressible as a set of

postconditions on all operations that may change the state of objects of the class. While invariants are easier to use on a specification level, the translation into equivalent postconditions seems beneficial for providing efficient implementations of constraints. If constraints can be effectively checked with little overhead, it is more probable that they are not only used for the specification of a system but also in the implementation. As a result, constraints may not only contribute to the correctness of a design but also to the correctness of an implementation.

- The precise meaning of inheritance of OCL invariants and pre- and postconditions has yet to be defined. Inheritance of constraints strongly depends on how subtyping and subclassing is defined in UML models. It seems that in UML these concepts are intentionally underspecified in order to be open to different object-oriented approaches.

- Extensibility is a major goal of UML. As OCL is used more and more in different contexts, it also seems desirable to have extension mechanisms for the constraint language of UML. For example, the set of operations is currently fixed. So far, it has been extended in each revision of the standard. A possible solution would be to provide some kind of library concept, where users can define their own operations. A simple approach which has already been used extensively for the well-formedness rules in the UML metamodel is the definition of "additional operations" in terms of OCL expressions. A more powerful extension mechanism could be achieved by the introduction of higher-order functions. In particular, the class of iterate-based expressions can be easily extended without changing the language if the argument of an iterate expression is passed as function. While the language core could thus be reduced, the cost is an increased complexity of the type system.

We have seen that OCL is an important part of the Unified Modeling Language with many interesting applications. OCL offers the potential of adding rigor and preciseness to software designs. In our view, much of the power of UML originates from the combination of the visual language of UML with the formal constraint language OCL.

Appendix A

Syntax of USE Specifications

This section presents grammars for the specification, expression, and command languages of USE. We use a BNF-like notation with the following conventions. Terminal symbols are strings that are enclosed in single quotes and printed in bold. All other strings denote nonterminal symbols. The nonterminal of the first production is the start symbol of the grammar. Elements of a sequence are separated with blanks, alternatives are separated by a "|" symbol. An optional element is enclosed in brackets "[" and "]", zero or more occurrences of an element are expressed by braces "{" and "}". Parentheses "(" and ")" are used to group elements in the grammar.

Nonterminal symbols of the form "xId" or "xIdList" have the same definition as the nonterminals "id" and "idList", respectively. In these cases, the prefix x is only used for giving additional information about the meaning of the expected identifier. Checking that an identifier really conforms to the expected kind is outside the scope of the grammar formalism and usually requires context information.

The grammar in Section A.1 describes the syntax for specifying structural aspects of a model. The nonterminal "expression" is defined in the grammar for expressions in Section A.2. It is based on the OCL grammar given in [OMG99b]. Some changes have been made to make the grammar stricter and compliant to the OCL syntax defined in Chapter 5. For example, the rules for "iterateExpression" and "typeExpression" were introduced, because in our approach these constructs are not operations. These expressions may have special arguments such as element variable declarations or type names. The rule "undefinedLiteral" enables the explicit use of undefined values in expressions. The rule "emptyCollectionLiteral" is specific to the USE implementation in that it allows a simplified type checking algorithm for empty collection literals.

A.1 Grammar for Object Models

model	::=	'**model**' modelId { enumTypeDefinition } { classDefinition | associationDefinition | '**constraints**' { invariant | prePost } }
enumTypeDefinition	::=	'**enum**' typeId '**{**' literalIdList '**}**'
classDefinition	::=	['**abstract**'] '**class**' classId [parents] [attributes] [operations] [constraints] '**end**'
parents	::=	'**<**' classIdList
attributes	::=	'**attributes**' { attributeDefinition }
attributeDefinition	::=	attrId '**:**' type
operations	::=	'**operations**' { operationDefinition }
operationDefinition	::=	opId paramList ['**:**' type ['**=**' expression]] { prePostClause }
constraints	::=	'**constraints**' { invariantClause }
associationDefinition	::=	associationKind assocId '**between**' associationEnd associationEnd { associationEnd } '**end**'
associationKind	::=	'**association**' | '**aggregation**' | '**composition**'
associationEnd	::=	aendId '**[**' multiplicity '**]**' ['**role**' roleId] ['**ordered**']
multiplicity	::=	multiplicityRange { '**,**' multiplicityRange }
multiplicityRange	::=	multiplicitySpec ['**..**' multiplicitySpec]
multiplicitySpec	::=	int | '*****'
invariant	::=	invariantContext invariantClause { invariantClause }
invariantContext	::=	'**context**' [varId '**:**'] simpleType
invariantClause	::=	'**inv**' [invId] '**:**' expression
prePost	::=	prePostContext prePostClause { prePostClause }
prePostContext	::=	'**context**' [classId '**::**'] opId paramList ['**:**' type]
prePostClause	::=	('**pre**' | '**post**') [prePostId] '**:**' expression
paramList	::=	'**(**' [variableDeclaration { '**,**' variableDeclaration }] '**)**'
variableDeclaration	::=	varId '**:**' type
type	::=	simpleType | collectionType
simpleType	::=	typeId
collectionType	::=	('**Collection**' | '**Set**' | '**Sequence**' | '**Bag**') '**(**' type '**)**'
id	::=	('**a**'..'**z**' | '**A**'..'**Z**' | '_') { '**a**'..'**z**' | '**A**'..'**Z**' | '_' | '**0**'..'**9**' }
idList	::=	id { '**,**' id }
int	::=	'**0**'..'**9**' { '**0**'..'**9**' }
real	::=	int ['**.**' int] [('**e**' | '**E**') ['**+**' | '**-**'] int]
string	::=	'**'** { charactersOrEscapeCodes } '**'**

A.2 Grammar for Expressions

expression	::=	{ '**let**' varId ['**:**' type] '**=**' expression '**in**' } conditionalImpliesExpression
conditionalImpliesExpression	::=	conditionalOrExpression { '**implies**' conditionalOrExpression }
conditionalOrExpression	::=	conditionalXOrExpression { '**or**' conditionalXOrExpression }
conditionalXOrExpression	::=	conditionalAndExpression { '**xor**' conditionalAndExpression }
conditionalAndExpression	::=	equalityExpression { '**and**' equalityExpression }
equalityExpression	::=	relationalExpression { ('**=**' | '**<>**') relationalExpression }
relationalExpression	::=	additiveExpression { ('**<**' | '**>**' | '**<=**' | '**>=**') additiveExpression }
additiveExpression	::=	multiplicativeExpression { ('**+**' | '**-**') multiplicativeExpression }
multiplicativeExpression	::=	unaryExpression { ('*****' | '**/**' | '**div**') unaryExpression }
unaryExpression	::=	postfixExpression | ('**not**' | '**-**' | '**+**') unaryExpression
postfixExpression	::=	primaryExpression { ('**.**' | '**->**') propertyCall }
primaryExpression	::=	literal | propertyCall | '**(**' expression '**)**' | ifExpression | classId '**.**' '**allInstances**'
propertyCall	::=	queryExpression | iterateExpression | operationExpression | typeExpression
queryExpression	::=	('**select**' | '**reject**' | '**collect**' | '**exists**' | '**forAll**' | '**isUnique**' | '**sortedBy**') '**(**' [elemVarsDeclaration '**|**'] expression '**)**'
iterateExpression	::=	'**iterate**' '**(**' elemVarsDeclaration '**;**' variableInitialization '**|**' expression '**)**'
operationExpression	::=	opId ['**@**' '**pre**'] ['**(**' [expression { '**,**' expression }] '**)**']
typeExpression	::=	('**oclAsType**' | '**oclIsKindOf**' | '**oclIsTypeOf**') '**(**' type '**)**'
elemVarsDeclaration	::=	varIdList ['**:**' type]
variableInitialization	::=	varId '**:**' type '**=**' expression
ifExpression	::=	'**if**' expression '**then**' expression '**else**' expression '**endif**'
literal	::=	'**true**' | '**false**' | int | real | string | '**#**' enumId | collectionLiteral | emptyCollectionLiteral | undefinedLiteral
collectionLiteral	::=	('**Set**' | '**Sequence**' | '**Bag**') '**{**' collectionItem { '**,**' collectionItem } '**}**'
collectionItem	::=	expression ['**..**' expression]
emptyCollectionLiteral	::=	'**oclEmpty**' '**(**' collectionType '**)**'
undefinedLiteral	::=	'**oclUndefined**' '**(**' type '**)**'

ProjectHeader: work 0.215 Mon, 20 Aug 2001 16:30:46 +0200 mr

A.3 Grammar for State Manipulation Commands

This grammar documents the syntax of primitive commands available for manipulating system states in the USE tool. There are commands for creating and destroying objects, for inserting and deleting links between objects, setting attribute values of objects, and simulating entry and exit from operation calls. Section B.2 gives an example for a sequence of state manipulation commands with concrete arguments.

```
command       ::=  createCmd
                 | destroyCmd
                 | insertCmd
                 | deleteCmd
                 | setCmd
                 | opEnterCmd
                 | opExitCmd

createCmd     ::=  'create' objectIdList ':' simpleType
destroyCmd    ::=  'destroy' objectIdList
insertCmd     ::=  'insert' '(' objectIdList ')' 'into' associationId
deleteCmd     ::=  'delete' '(' objectIdList ')' 'from' associationId
setCmd        ::=  'set' objectId '.' attributeId '=' expression
opEnterCmd    ::=  'openter' expression opId
                   '(' [ expression { ',' expression } ] ')'
opExitCmd     ::=  'opexit' [ expression ]
```

Appendix B

Specification of the Case Study

The following is a complete specification of the example case study presented on page 11. The specification is suitable for validation with the USE tool described in Chapter 7. Section B.1 contains the USE specification. Section B.2 lists commands that can be used to create and manipulate states conforming to the specification. The state resulting from these commands is shown in a screenshot in Section B.3.

B.1 USE Specification

```
model CarRental

------------------------------------
-- Classes
------------------------------------

abstract class Person
attributes
  firstname : String
  lastname : String
  age : Integer
  isMarried : Boolean
  email : Set(String)
operations
  -- Produce a full name, e.g. 'Mr. Frank Black'.
  -- This is an operation without side effects,
  -- the method body is given as an OCL expression.
  fullname(prefix : String) : String =
    prefix.concat(' ').concat(firstname)
      .concat(' ').concat(lastname)
end
```

```
class Customer < Person
attributes
  address : String
end

class Employee < Person
attributes
  salary : Real
operations
  -- This operation has side effects,
  -- the method body is left unspecified.
  raiseSalary(amount : Real) : Real
end

class Branch
attributes
  location : String
operations
  -- Query all rentals for a given day
  rentalsForDay(day : String) : Set(Rental) =
    rental->select(r : Rental |
      r.fromDay <= day and day <= r.untilDay)
end

class Rental
attributes
  fromDay : String
  untilDay : String
end

class CarGroup
attributes
  kind : String -- compact, intermediate, luxury
operations
  -- Transitive closure of higher grade cars
  allHigher() : Set(CarGroup) =
    if higher->isEmpty() then
      Set{self}
    else
      Set{self}->union(higher.allHigher())
    endif

  -- Transitive closure of lower grade cars
  allLower() : Set(CarGroup) =
    if lower->isEmpty() then
      Set{self}
    else
      Set{self}->union(lower.allLower())
    endif

  isEqualOrBetterThan(other : CarGroup) : Boolean =
    self.allLower()->includes(other)
end
```

ProjectHeader: work 0.215 Mon, 20 Aug 2001 16:30:46 +0200 mr

```
class Car
attributes
  id : String
operations
  description() : String =
    id.concat(' of group ').concat(carGroup.kind)
end

class ServiceDepot
attributes
  location : String
end

class Check
attributes
  description : String
end

------------------------------------
-- Associations
------------------------------------

association Management between
  Employee[1] role manager
  Branch[0..1] role managedBranch
end

association Employment between
  Employee[*] role employee
  Branch[1] role employer
end

association Fleet between
  Branch[1]
  Car[*]
end

association Offers between
  Branch[*]
  CarGroup[*]
end

association Classification between
  CarGroup[1]
  Car[*]
end

association Booking between
  Rental[*]
  Customer[1]
end

association Provider between
```

```
  Rental[*]
  Branch[1]
end

association Reservation between
  Rental[*]
  CarGroup[1]
end

association Assignment between
  Rental[0..1]
  Car[0..1]
end

association Quality between
  CarGroup[0..1] role lower
  CarGroup[0..1] role higher
end

association Maintenance between
  ServiceDepot[0..1]
  Check[*]
  Car[*]
end

------------------------------------
-- Constraints
------------------------------------

constraints

context Person
-- [1] The age attribute of persons is greater than zero.
inv Person1:
  age > 0

-- [2] Both names must be defined.
inv Person2:
  firstname.isDefined() and lastname.isDefined()


context Branch
-- [1] Each manager is also an employee of the same branch.
inv Branch1:
  self.employee->includes(self.manager)

-- [2] Managers get a higher salary than employees.
inv Branch2:
  self.employee->forAll(e |
    e <> self.manager implies self.manager.salary > e.salary)


context CarGroup
-- [1] The CarGroups association is not reflexive.
```

```
inv CarGroup1:
  higher <> self and lower <> self

-- [2] The CarGroups association is anti-symmetric.
inv CarGroup2:
  higher.higher <> self and lower.lower <> self

-- [3] There is exactly one CarGroup with lowest grade.
inv CarGroup3:
  CarGroup.allInstances->select(cg |
    cg.lower->isEmpty())->size() = 1

-- [4] There is exactly one CarGroup with highest grade.
inv CarGroup4:
  CarGroup.allInstances->select(cg |
    cg.higher->isEmpty())->size() = 1

-- [5] All CarGroup objects are connected.
inv CarGroup5:
  CarGroup.allInstances->iterate(cg;
    s : Set(CarGroup) = oclEmpty(Set(CarGroup)) |
    s->including(cg.higher)->including(cg.lower))
   ->excluding(oclUndefined(CarGroup)) = CarGroup.allInstances


context Car
-- [1] A car may not be assigned to a maintenance and to a
-- rental at the same time.
inv Car1:
  rental->isEmpty() or serviceDepot->isEmpty()

-- [2] A maintenance is done in only one service depot (this
-- cannot be expressed with multiplicities on ternary
-- associations).
inv Car2:
  serviceDepot->size() <= 1


context Rental
-- [1] A reserved car group must be offered by the branch
-- providing the reservation
inv Rental1:
  self.branch.carGroup->includes(self.carGroup)

-- [2] Only a car of the requested car group or a higher one
-- ("upgrading") may be assigned to a rental
inv Rental2:
  car->notEmpty() implies
    car.carGroup.isEqualOrBetterThan(carGroup)


context Employee::raiseSalary(amount : Real) : Real
  -- If the amount is positive, raise the salary
  -- by the given amount and return the new salary
```

```
  pre:  amount > 0
  post: self.salary = self.salary@pre + amount
          and result = self.salary
```

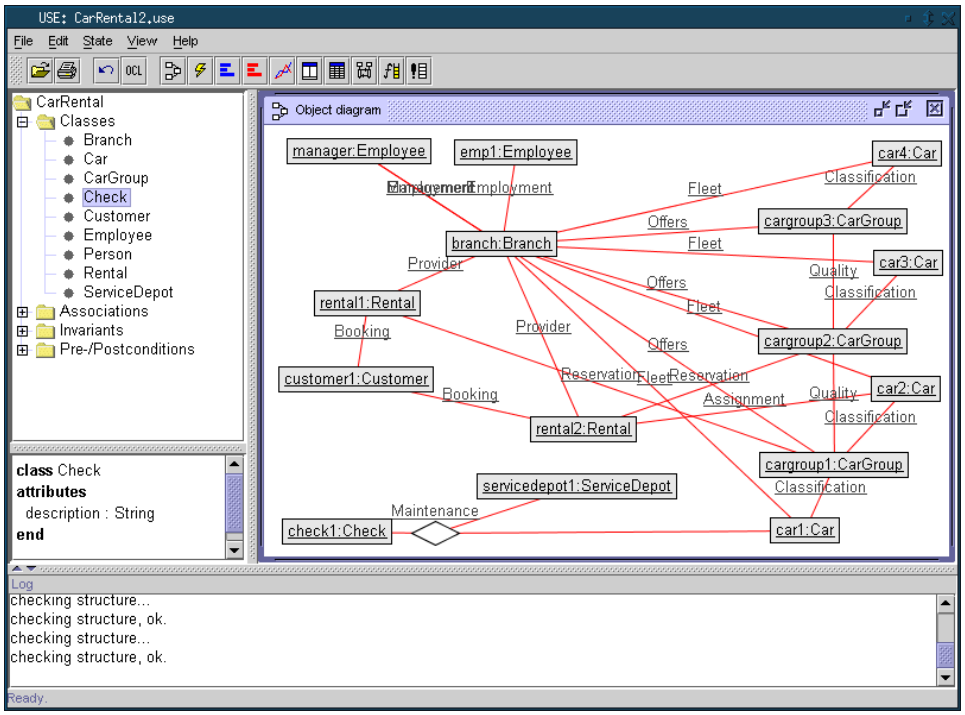
B.2 State Manipulation

```
-- Branch objects
!create branch : Branch
!set branch.location = 'Bremen'

-- Car objects
!create car1, car2, car3, car4 : Car
!set car1.id = 'Red car'
!set car2.id = 'Blue car'
!set car3.id = 'Silver car'
!set car4.id = 'Gold car'
!insert (branch, car1) into Fleet
!insert (branch, car2) into Fleet
!insert (branch, car3) into Fleet
!insert (branch, car4) into Fleet

-- CarGroup objects
!create cargroup1, cargroup2, cargroup3 : CarGroup
!set cargroup1.kind = 'compact'
!set cargroup2.kind = 'intermediate'
!set cargroup3.kind = 'luxury'

-- Offers links
!insert (branch, cargroup1) into Offers
!insert (branch, cargroup2) into Offers
!insert (branch, cargroup3) into Offers

-- Quality
!insert(cargroup1, cargroup2) into Quality
!insert(cargroup2, cargroup3) into Quality

-- Classification
!insert(cargroup1, car1) into Classification
!insert(cargroup1, car2) into Classification
!insert(cargroup2, car3) into Classification
!insert(cargroup3, car4) into Classification

-- Customer objects
!create customer1 : Customer
!set customer1.firstname = 'Frank'
!set customer1.lastname = 'Black'
!set customer1.age = 36
!set customer1.address = 'Hamburg'

-- Employee (manager) objects
!create manager : Employee
```

ProjectHeader: work 0.215 Mon, 20 Aug 2001 16:30:46 +0200 mr

```
!set manager.firstname = 'Joe'
!set manager.lastname = 'White'
!set manager.age = 48
!set manager.isMarried = true
!set manager.email = Set{'joe@branch.com'}
!set manager.salary = 4000.0

!insert(manager, branch) into Management
!insert(manager, branch) into Employment

-- Employee objects
!create emp1 : Employee
!set emp1.firstname = 'Jack'
!set emp1.lastname = 'Green'
!set emp1.age = 28
!set emp1.isMarried = false
!set emp1.email = Set{'jack@branch.com'}
!set emp1.salary = 3000.0

!insert(emp1, branch) into Employment

-- Rental objects
!create rental1 : Rental
!insert (rental1, branch) into Provider
!insert (rental1, customer1) into Booking
!insert (rental1, cargroup1) into Reservation

!create rental2 : Rental
!insert (rental2, branch) into Provider
!insert (rental2, customer1) into Booking
!insert (rental2, cargroup2) into Reservation
!insert (rental2, car2) into Assignment

-- ServiceDepot objects
!create servicedepot1 : ServiceDepot

-- Check objects
!create check1 : Check
!set check1.description = 'Brakes'
!insert (servicedepot1, check1, car1) into Maintenance
```

B.3 Example State



Appendix C

Specification of the OCL Metamodel

The USE specification in Section C.1 describes the OCL metamodel presented in Chapter 6. In that chapter, example states representing OCL types and expressions were given. These can be created with the commands given in Section C.2 and Section C.3, respectively.

C.1 USE Specification

```
model OCLmetamodel

class ModelElement -- from Core
attributes
  name : String
end

class Constraint -- from Core
end

class PreCondition < Constraint
end

class PostCondition < Constraint
end

class Invariant < Constraint
end

class Guard < Constraint
end

class Expression -- from Data Types
```

```
end

class BooleanExpression < Expression -- from Data Types
end

association Constraint_ModelElement between
  Constraint[1..*]
  ModelElement[1..*] role constrainedElement ordered
end

composition Constraint_BooleanExpression between
  Constraint[0..1]
  BooleanExpression[1] role body
end

composition Expression_OclExpression between
  Expression[0..1]
  OclExpression[0..1] role body
end

------------------------------------
-- Package: Types
------------------------------------

abstract class Type
attributes
  name : String
end

abstract class BasicType < Type
end

class IntegerType < BasicType
end

class RealType < BasicType
end

class StringType < BasicType
end

class BooleanType < BasicType
end

class ObjectType < Type
end

class Classifier < ModelElement -- from Core
end

class Class < Classifier -- from Core
end

class EnumType < Type
```

```
    --constraints
    --FIXME:  self.literal->isUnique(name)
end

class EnumLiteral
attributes
  name : String
constraints
  inv: self.name.size > 0
end

class OclAnyType < Type
end

class OclTypeType < Type
end

class CollectionType < Type
end

class SetType < CollectionType
end

class SequenceType < CollectionType
end

class BagType < CollectionType
end


association Conforms between
  Type[*] role subtype
  Type[*] role supertype
end

association CollectionType_Type between
  CollectionType[*]
  Type[1] role elementType
end

aggregation ObjectType_Classifier between
  ObjectType[0..1]
  Classifier[1]
end

aggregation EnumType_EnumLiteral between
  EnumType[0..1]
  EnumLiteral[1..*]
end

constraints

context IntegerType inv:
  self.name = 'IntegerType'
```

```
------------------------------------
-- Package: Values
------------------------------------

abstract class Value
end

abstract class BasicValue < Value
end

class IntegerValue < BasicValue
attributes
  val : Integer
end

class RealValue < BasicValue
attributes
  val : Real
end

class StringValue < BasicValue
attributes
  val : String
end

class BooleanValue < BasicValue
attributes
  val : Boolean
end

class ObjectValue < Value
end

class Object < ModelElement -- from Common Behavior
end

class EnumValue < Value
end

class UndefinedValue < Value
end

abstract class CollectionValue < Value
end

class SequenceValue < CollectionValue
end

class BagValue < CollectionValue
end

class SetValue < CollectionValue
end
```

ProjectHeader: work 0.215 Mon, 20 Aug 2001 16:30:46 +0200 mr

```
class BagOccurrence
attributes
  count : Integer
end

class SequenceOccurrence
attributes
  index : Integer
end


association Value_Type between
  Value[*]
  Type[1]
end

association ObjectValue_Object between
  ObjectValue[0..1]
  Object[1]
end

aggregation EnumValue_EnumLiteral between
  EnumValue[0..1]
  EnumLiteral[1]
end

aggregation SetValue_Value between
  SetValue[*]
  Value[*] role elements
end

aggregation BagValue_BagOccurrence between
  BagValue[1]
  BagOccurrence[*]
end

association BagOccurrence_Value between
  BagOccurrence[*]
  Value[1] role element
end

aggregation SequenceValue_SequenceOccurrence between
  SequenceValue[1]
  SequenceOccurrence[*]
end

association SequenceOccurrence_Value between
  SequenceOccurrence[*]
  Value[1] role element
end

------------------------------------
-- Package: Expressions
```

```
-----------------------------------

abstract class OclExpression
end

class Context
end

class VariableDeclaration
attributes
  var : String
end

class VariableInitialization
end

class VariableExp < OclExpression
attributes
  var : String
end

class LetExp < OclExpression
attributes
  var : String
end

abstract class OperationExp < OclExpression
end

class IfExp < OclExpression
end

class TypingExp < OclExpression
end

abstract class QueryExp < OclExpression
end

class ForAllExp < QueryExp
end

class ExistsExp < QueryExp
end

class SelectExp < QueryExp
end

class RejectExp < QueryExp
end

class CollectExp < QueryExp
end

class IsUniqueExp < QueryExp
```

```
end

class SortedByExp < QueryExp
end

class IterateExp < QueryExp
end

class PropertyOperation < OperationExp
attributes
  isMarkedPre : Boolean
end

class AttributeExp < PropertyOperation
end

class Attribute < ModelElement -- from Core
end

class NavigationExp < PropertyOperation
end

class AssociationEnd < ModelElement -- from Core
end

class ClassifierOperation  < PropertyOperation
end

class Operation < ModelElement -- from Core
end

class OclOperation < OperationExp
attributes
  name : String
end

abstract class ConstExp < OperationExp
end

class IntegerConstExp < ConstExp
attributes
  value : Integer
end

association Context_Classifier between
  Context[*]
  Classifier[0..1]
end

association Context_Operation between
  Context[*]
  Operation[0..1]
end
```

```
association Context_OclExpression between
  Context[0..1]
  OclExpression[*]
end

association Context_VariableDeclaration between
  Context[0..1]
  VariableDeclaration[*]
end

aggregation VariableDeclaration_Type between
  VariableDeclaration[*]
  Type[1] role varType
end

association OclExpression_Type between
  OclExpression[*]
  Type[1] role resultType
end

association OclExpression_Value between
  OclExpression[*]
  Value[1] role result
end

aggregation LetExp_OclExpression between
  LetExp[0..1]
  OclExpression[1] role in_
end

aggregation LetExp_VariableInitialization between
  LetExp[0..1]
  VariableInitialization[1]
end

aggregation IfExp_OclExpression_Condition between
  IfExp[0..1] role conditionIf
  OclExpression[1] role condition
end

aggregation IfExp_OclExpression_Then between
  IfExp[0..1] role thenIf
  OclExpression[1] role then_
end

aggregation IfExp_OclExpression_Else between
  IfExp[0..1] role elseIf
  OclExpression[1] role else_
end

aggregation TypingExp_OclExpression between
  TypingExp[0..1]
  OclExpression[1] role source
end
```

ProjectHeader: work 0.215 Mon, 20 Aug 2001 16:30:46 +0200 mr

```
aggregation TypingExp_Type between
  TypingExp[0..1]
  Type[1] role argument
end

aggregation QueryExp_OclExpression_Source between
  QueryExp[0..1]
  OclExpression[1] role source
end

aggregation QueryExp_OclExpression_Argument between
  QueryExp[0..1] role owner
  OclExpression[1] role argument
end

aggregation QueryExp_VariableDeclaration between
  QueryExp[0..1]
  VariableDeclaration[0..1]
end

aggregation VariableInitialization_VariableDeclaration between
  VariableInitialization[0..1]
  VariableDeclaration[1]
end

aggregation VariableInitialization_OclExpression between
  VariableInitialization[0..1]
  OclExpression[1] role initExpression
end

aggregation IterateExp_VariableInitialization between
  IterateExp[1]
  VariableInitialization[1] role varInit
end

aggregation PropertyOperation_OclExpression between
  PropertyOperation[*]
  OclExpression[1] role source
end

aggregation OclOperation_OclExpression between
  OclOperation[*]
  OclExpression[*] role arguments ordered
end

aggregation AttributeExp_Attribute between
  AttributeExp[*]
  Attribute[1]
end

aggregation NavigationExp_AssociationEnd between
  NavigationExp[*]
  AssociationEnd[1]
```

```
end

aggregation NavigationExp_OclExpression between
  NavigationExp[*]
  OclExpression[*] role qualifierValues
end

aggregation ClassifierOperation_OclExpression between
  ClassifierOperation[*]
  OclExpression[*] role arguments ordered
end

aggregation ClassifierOperation_Operation between
  ClassifierOperation[*]
  Operation[1]
end

------------------------------------
-- Well-formedness rules
------------------------------------

constraints

context Constraint

-- [1] When a BooleanExpression is used in a Constraint and
-- it is defined by an OclExpression, the Type of the
-- OclExpression must be an instance of BooleanType.

inv Constraint1:
  let b = self.body.body in
    b.isDefined() implies b.resultType.oclIsKindOf(BooleanType)

context PostCondition

-- [1] A PostCondition may only be attached to an Operation.

inv PostCondition1:
  self.constrainedElement->forAll(me : ModelElement |
    me.oclIsKindOf(Operation))

context Type

-- [1] OclAnyType is the supertype of all Types except for
-- the collection types.

inv Type1:
  not self.oclIsKindOf(CollectionType)
    implies self.supertype->exists(t : Type |
      t.oclIsKindOf(OclAnyType))

-- [2] Type conformance (represented by the association
-- Conforms) is transitive.
```

```
inv Type2:
  Type.allInstances->forAll(t1, t3 : Type |
    Type.allInstances->exists(t2 : Type |
      (t1.subtype->includes(t2) and t2.subtype->includes(t3))
        implies t1.subtype->includes(t3)))

context EnumType

-- [1] All EnumerationLiterals of an EnumType are distinct.

inv EnumType1:
  self.enumLiteral->isUnique(el : EnumLiteral | el.name)

context SetType

-- [1] A set type Set(T1) conforms to a type Set(T2) if its
-- element type T1 conforms to T2.

inv SetType1:
  SetType.allInstances->forAll(s1, s2 : SetType |
    s1.elementType.supertype->includes(s2.elementType)
      implies s1.supertype->includes(s2))

context QueryExp

-- [1] The source expression of a QueryExp must have a
-- collection type.

inv QueryExp1:
  self.source.resultType.oclIsKindOf(CollectionType)
```

C.2 Commands for Creating a Type

The following commands create the instances shown in the object diagram in Figure 6.7 on page 123. The resulting state can thus be checked for validity with the USE tool.

```
-----------------------------------
-- Type Example
-----------------------------------

-- Person class object
!create personClass : Class
!set personClass.name = 'Person'

-- Type of class Person
!create personClassType : ObjectType
!insert (personClassType, personClass)
        into ObjectType_Classifier
```

```
-- Type of Set(Person)
!create personSetType : SetType
!insert (personSetType, personClassType) into CollectionType_Type

-- Employee class object
!create employeeClass : Class
!set employeeClass.name = 'Employee'

-- Type of class Employee
!create employeeClassType : ObjectType
!insert (employeeClassType, employeeClass)
        into ObjectType_Classifier

-- Type of Set(Employee)
!create employeeSetType : SetType
!insert (employeeSetType, employeeClassType)
        into CollectionType_Type

-- Type of Person is a supertype of Employee type
!insert (employeeClassType, personClassType) into Conforms

-- Set types conform
!insert (employeeSetType, personSetType) into Conforms
```

C.3 Commands for Creating an Expression

The following commands create the instances shown in the object diagram in Figure 6.12 on page 130.

```
-------------------------------------------------
-- Expression example:
-- self.employees->select(p : Person | p.age > 45)
-------------------------------------------------

!create selectExp : SelectExp

-- Source of select expression
!create navigationExp : NavigationExp
!insert (selectExp, navigationExp)
        into QueryExp_OclExpression_Source

!create varSelfExp : VariableExp
!set varSelfExp.var = 'self'
!insert (navigationExp, varSelfExp)
        into PropertyOperation_OclExpression

!create associationEnd : AssociationEnd
!set associationEnd.name = 'employees'
!insert (navigationExp, associationEnd)
```

```
        into NavigationExp_AssociationEnd

-- Argument of select expression
!create greaterOp : OclOperation
!set greaterOp.name = '>'
!insert (selectExp, greaterOp)
        into QueryExp_OclExpression_Argument

-- Subexpression 'p.age'
!create attributeExp : AttributeExp
!insert (greaterOp, attributeExp)
        into OclOperation_OclExpression
!create varPExp : VariableExp
!set varPExp.var = 'p'
!insert (attributeExp, varPExp)
        into PropertyOperation_OclExpression
!create ageAttribute : Attribute
!set ageAttribute.name = 'age'
!insert (attributeExp, ageAttribute)
        into AttributeExp_Attribute

-- Subexpression '45'
!create int45 : IntegerConstExp
!set int45.value = 45
!insert (greaterOp, int45)
        into OclOperation_OclExpression
```

Bibliography

[Aag98] J. Ø. Aagedal. Towards an ODP-compliant object definition language with QoS-support. In T. Plagemann and V. Goebel, editors, *Proceedings 5th International Workshop, IDMS'98, Oslo, Norway, September*, volume 1483 of *LNCS*, pages 183–194. Springer, 1998. 19

[ABLV81] P. Atzeni, C. Batini, M. Lenzerini, and F. Villanelli. INCOD: A System for Conceptual Design of Data and Transactions in the Entity-Relationship Model. In P. P. Chen, editor, *Proc. 2nd Int. Conf. on the Entity-Relationship Approach (ER'81)*, pages 375–410. North Holland, 1981. 17, 30

[AEF+99] E. Astesiano, A. Evans, R. France, G. Geniloud, M. Gogolla, B. Henderson-Sellers, J. Howse, H. Hussmann, S. Iida, S. Kent, A. Le Guennec, T. Mens, R. Mitchell, O. Radfelder, G. Reggio, M. Richters, B. Rumpe, P. Stevens, K. van den Berg, P. van den Broek, and R. Wieringa. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *ECOOP'99 Workshop Reader*, pages 33–56. Springer, Berlin, LNCS 1743, 1999. 2, 4

[AFGP96] A. Artale, E. Franconi, N. Guarino, and L. Pazzi. Part-whole relations in object-centered systems: An overview. *Data & Knowledge Engineering*, 20(3):347–383, November 1996. 41

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995. 45, 46

[AMDK98] L. Andrade, A. Moreira, A. Deshpande, and S. Kent, editors. *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?* 1998. 1

[Ara98] J. Araújo. Formalizing sequence diagrams. In L. Andrade, A. Moreira, A. Deshpande, and S. Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?* 1998. 2

[Baa00] T. Baar. Experiences with the UML/OCL-approach in practice and strategies to overcome deficiencies. In Net.ObjectDays-Forum, editor, *Proc. Net.ObjectDays2000, Erfurt, Germany*, pages 192–201. October 2000. 19, 21

[Bee90] C. Beeri. A Formal Approach to Object-Oriented Databases. *Data & Knowledge Engineering*, 5(4):353–382, 1990. 22

[BEE00] R. Bardohl, H. Ehrig, and C. Ermel. Generic description, behavior and animation of visual modeling languages. In P. A. Ng, editor, *Proc. Fifth International Conference on Integrated Design and Process Technology (IDPT'2000), June 2000, Dallas, Texas*. 2000. 134

[BGH+97] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Towards a precise semantics for object-oriented modeling techniques. In H. Kilov and B. Rumpe, editors, *Proceedings ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques*, pages 53–59. Technische Universität München, TUM-I9725, 1997. 2

[BH00] T. Baar and R. Hähnle. An integrated metamodel for OCL types. In R. France, B. Rumpe, J.-M. Bruel, A. Moreira, J. Whittle, and I. Ober, editors, *Proc. OOPSLA 2000, Workshop Refactoring the UML: In Search of the Core, Minneapolis, Minnesota, USA, 2000.* 2000. 3, 75, 115

[BHH+97] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the Unified Modeling Language. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 – Object-Oriented Programming, 11th European Conference*, volume 1241 of *LNCS*, pages 344–366. Springer, 1997. 7

[BHS99] F. Barbier and B. Henderson-Sellers. Object metamodelling of the whole-part relationship. In C. Mingins, editor, *Proceedings of TOOLS Pacific 1999*. IEEE Computer Society, 1999. 41

[BHSOG01] F. Barbier, B. Henderson-Sellers, A. L. Opdahl, and M. Gogolla. The whole-part relationship in the Unified Modeling Language: A new approach. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, chapter 12, pages 185–209. Idea Publishing Group, 2001. 41

[BHSS00] T. Baar, R. Hähnle, T. Sattler, and P. H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In

K. Mehlhorn and G. Snelting, editors, *Informatik 2000, 30. Jahrestagung der Gesellschaft für Informatik*, pages 389–404. September 2000. 19

[BHTW99] M. Bidoit, R. Hennicker, F. Tort, and M. Wirsing. Correct realization of interface constraints with OCL. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 399–415. Springer, 1999. 3

[BKPPT00] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 294–308. Springer, 2000. 2, 10

[BM99] J. Bézivin and P.-A. Muller, editors. *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*. Springer, 1999. 1

[Bod00] M. Bodenmüller. The OCL metamodel and the UML_OCL package. In J. Warmer and T. Clark, editors, *Proc. UML 2.0 - The Future of the UML Object Constraint Language (OCL), UML 2000 Workshop, York*. 2000. 115

[Boh01] J. Bohling. *Generierung von Snapshots zur Validation von UML-Klassendiagrammen*. Diplomarbeit, Universität Bremen, Fachbereich Informatik, 2001. 152

[Bol00] BoldSoft. Modelrun, 2000. Internet: `http://www.boldsoft.com/products/modelrun/index.html`. 20, 134

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1994. 1, 5

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Communications of the ACM*, 34(10):64–77, October 1991. 25

[BP97] M. Blaha and W. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, 1997. 27

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998. 1, 5

[CB00] R. G. G. Cattell and D. K. Berry, editors. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann Publishers, Inc., 2000. 17, 25

[CD94] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice Hall, New York, 1994. 27

[CEK01] T. Clark, A. Evans, and S. Kent. The metamodelling language calculus: Foundation semantics for UML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6. 2001 Proceedings*, volume 2029 of *LNCS*, pages 17–31. Springer, 2001. 3, 7

[Che76] P. P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Trans. on Database Systems*, 1(1):9–36, 1976. 17, 30

[CHMW96] M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams. PESTO : An integrated query/browser for object databases. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 203–214. Morgan Kaufmann, 1996. 14

[CK94] S. R. Chidamber and C. F. Kemerer. A metric suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994. 151

[CKM+99a] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The Amsterdam manifesto on OCL. Technical Report TUM-I9925, Technische Universität München, December 1999. 21, 58, 74

[CKM+99b] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. Defining UML family members using prefaces. In C. Mingins, editor, *Proceedings of TOOLS Pacific 1999.* IEEE Computer Society, 1999. 156

[CKM+99c] S. Cook, A. Kleppe, R. Mitchell, J. Warmer, and A. Wills. Defining the context of OCL expressions. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort*

Collins, CO, USA, October 28-30. 1999, Proceedings, volume 1723 of *LNCS*, pages 372–383. Springer, 1999. 96

[CKW00a] T. Clark, S. Kent, and J. Warmer, editors. *Proc. of the Workshop on Expressing and Reasoning about Constraints in UML, 13 - 14 March 2000, University of Kent*. 2000. 1

[CKW+00b] T. Clark, S. Kent, J. Warmer, et al. OCL Semantics FAQ, Workshop on the Object Constraint Language (OCL) Computing Laboratory, University of Kent, Canterbury, UK, March 2000. Internet: http://www.cs.ukc.ac.uk/research/sse/oclws2k/index.html. 152, 156

[Cla99] T. Clark. Type checking UML static diagrams. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 503–517. Springer, 1999. 3, 95

[Coo00] S. Cook. The UML family: Profiles, prefaces and packages. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 255–264. Springer, 2000. 156

[CR99] P. Collet and R. Rousseau. Towards efficient support for executing the Object Constraint Language. In *Tools 30 – USA'99. Proceedings.* IEEE Computer Society, 1999. 134

[CT01] S. Conrad and K. Turowski. Temporal OCL: Meeting specification demands for business components. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, chapter 10, pages 151–166. Idea Publishing Group, 2001. 3

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985. 23, 45, 76, 82

[CW00] T. Clark and J. Warmer, editors. *Proc. UML 2.0 - The Future of the UML Object Constraint Language (OCL), UML 2000 Workshop, York*. 2000. 1

[Dat90] C. J. Date. *An Introduction to Database Systems – Vol. I.* Addison-Wesley, Readings (MA), 1990. 55

[DdB00] S. Dupuy and L. du Bousquet. A multi-formalism approach for the validation of UML models. *Formal Aspects of Computing*, 12(4):228–230, 2000. 134

[Deu91] O. Deux. The O2 System. *Communications of the ACM*, 34(10):34–48, 1991. 25

[DKR00] D. Distefano, J.-P. Katoen, and A. Rensink. On a temporal logic for object-based systems. In S. F. Smith and C. L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2000, September, 2000, Stanford, California, USA*. Kluwer Academic Publishers, 2000. 3

[dSNF79] C. S. dos Santos, E. J. Neuhold, and A. L. Furtado. A Data Type Approach to the Entity-Relationship Approach. In P. P. Chen, editor, *Entity-Relationship Approach to Systems Analysis and Design. Proc. 1st International Conference on the Entity-Relationship Approach. Los Angeles, California, USA*, pages 103–119. North Holland, 1979. 17, 30

[DW98] D. D'Souza and A. Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998. 27, 72

[EDS93] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing Systems as Object Communities. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. Theory and Practice of Software Development (TAPSOFT'93)*, pages 453–467. Springer, Berlin, LNCS 668, 1993. 85

[EHHS00] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000. 10

[EJDS94] H.-D. Ehrich, R. Jungclaus, G. Denker, and A. Sernadas. Object-oriented design of information systems: Theoretical foundations. In J. Paredaens and L. Tenenbaum, editors, *Advances in Database Systems, Implementations and Applications*, pages 201–218. Springer Verlag, Wien, CISM Courses and Lectures no. 347, 1994. 85

[EKS00] A. Evans, S. Kent, and B. Selic, editors. *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*. Springer, 2000. 1

[EKTW86] J. Eder, G. Kappel, A. M. Tjoa, and R. R. Wagner. BIER — The Behavior Integrated Entity-Relationship Approach. In S. Spaccapietra, editor, *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*, pages 147–166. North-Holland, 1986. 17, 30

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., 2 edition, 1994. 55, 83

[Eva98] A. Evans. Making UML precise. In L. Andrade, A. Moreira, A. Deshpande, and S. Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?* 1998. 2

[EWH85] R. Elmasri, J. Weeldreyer, and A. Hevner. The Category Concept: An Extension to the Entity Relationship Model. *Data & Knowledge Engineering*, 1:75–116, may 1985. 17, 30

[FBLPS97] 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)*. Chapman and Hall, London, 1997. 2

[Fin00] F. Finger. *Design and Implementation of a Modular OCL Compiler*. Diplomarbeit, Dresden University of Technology, Department of Computer Science, Software Engineering Group, Germany, March 2000. 20, 134

[FR99] R. France and B. Rumpe, editors. *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999. 1

[FRHS+99] R. France, B. Rumpe, B. Henderson-Sellers, J.-M. Bruel, and A. Moreira, editors. *Proc. OOPSLA Workshop "Rigorous Modeling and Analysis with the UML: Challenges and Limitations"*. Colorado State University, Fort Collins, Colorado, 1999. 1

[FS97] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997. 1, 5

[GH91] M. Gogolla and U. Hohenstein. Towards a Semantic View of an Extended Entity-Relationship Model. *ACM Trans. on Database Systems*, 16(3):369–416, 1991. 28

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995. 121

[GK98] J. Gil and S. Kent. Three dimensional software modeling. In *Forging New Links, Proceedings of the 1998 International Conference on Software Engineering, ICSE 98, April 19-25, 1998, Kyoto, Japan*. IEEE Computer Society, 1998. 10

[Gog94] M. Gogolla. *An Extended Entity-Relationship Model – Fundamentals and Pragmatics*, volume 767 of *LNCS*. Springer, Berlin, 1994. 2, 17, 28, 50, 55

[Gog98] M. Gogolla. UML for the impatient. Research Report 3/98, Universität Bremen, 1998. 9, 10

[Gog00] M. Gogolla. Graph transformations on the UML metamodel. In J. D. P. Rolim, A. Z. Broder, A. Corradini, R. Gorrieri, R. Heckel, J. Hromkovic, U. Vaccaro, and J. B. Wells, editors, *Proc. ICALP Workshop Graph Transformations and Visual Modeling Techniques (GVMT'2000)*, pages 359–371. Carleton Scientific, Waterloo, Ontario, Canada, 2000. 10

[GPP98] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In M. Broy, D. Coleman, T. S. E. Maibaum, and B. Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998. 2, 10

[GR96] M. Gogolla and M. Richters. An Object Specification Language Implementation with Web User Interface based on Tycoon. In H. Ehrig, F. von Henke, J. Meseguer, and M. Wirsing, editors, *Specification and Semantics*, pages 8–11. Dagstuhl-Seminar-Report Nr. 151, 1996. 14

[GR98a] M. Gogolla and M. Richters. Equivalence rules for UML class diagrams. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 87–96. 1998. 4

[GR98b] M. Gogolla and M. Richters. On combining semi-formal and formal object specification techniques. In F. Parisi-Presice, editor, *Recent trends in algebraic development techniques: 12th international workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997: selected papers*, volume 1376 of *LNCS*. Springer, 1998. 4

[GR98c] 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*, pages 109–121. Physica-Verlag, Heidelberg, 1998. 4, 17, 21, 28, 59

[GR99] M. Gogolla and M. Richters. Transformation rules for UML class diagrams. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 92–106. Springer, 1999. 41, 49, 83

[GR00] M. Gogolla and M. Richters. Definition von UML mit UML und OCL: Ein Überblick zum Stand der Technik. In M. Jeckle, B. Rumpe, A. Schürr, and A. Winter, editors, *Proc. 7. GROOM-Workshop "UML - Erweiterungen (Profile) und Konzepte der Metamodellierung"*. Universität Koblenz-Landau, Fachbereich Informatik, 2000. Auch: Softwaretechnik-Trends, 20:2, 2000, ISSN 0720-8928. 4

[GRKR00] M. Gogolla, O. Radfelder, R. Kollmann, and M. Richters. Analysing Atomic Dynamic UML Notions by Surfing through the UML Metamodel. In G. Reggio, A. Knapp, B. Rumpe, B. Selic, and R. Wieringa, editors, *Proc. UML'00 Workshop Dynamic Behaviour in UML Models*. TU München, 2000. 4

[GRR99a] M. Gogolla, O. Radfelder, and M. Richters. Towards three-dimensional animation of UML diagrams. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 489–502. Springer, 1999. 4, 10

[GRR99b] M. Gogolla, O. Radfelder, and M. Richters. A UML semantics FAQ - the view from bremen. In S. J. H. Kent, A. Evans, and B. Rumpe, editors, *Proc. ECOOP'99 Workshop UML Semantics FAQ*. University of Brighton, 1999. 4

[GW91] J. A. Goguen and D. Wolfram. On types and FOOPS. In R. Meersman, W. Kent, and S. Khosla, editors, *Proceedings of the IFIP Working Group 2.6 Working Conference on Database Semantics: Object Oriented Databases: Analysis, Design & Construction*, pages 1–22. International Federation for Information Processing, North-Holland, Amsterdam, 1991. 85

[Ham99] A. Hamie. Enhancing the Object Constraint Language for more expressive specifications. In *Proceedings Asia Pacific Software Engineering Conference (APSEC '99), December, 1999, Takamatsu, Japan*. IEEE Computer Society, 1999. 3

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. 9

[HCH+99] A. Hamie, F. Civello, J. Howse, S. Kent, and R. Mitchell. Reflections on the Object Constraint Language. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 162–172. Springer, 1999. 3, 21

[HDF00] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 278–293. Springer, 2000. 20, 99, 134

[Her95] R. Herzig. *Zur Spezifikation von Objektgesellschaften mit TROLL light*. VDI-Verlag, Düsseldorf, Reihe 10 der Fortschritt-Berichte, Nr. 336, 1995. (Dissertation, Naturwissenschaftliche Fakultät, Technische Universität Braunschweig, 1994). 55, 67, 85

[HHK98a] A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proceedings 5th Asia Pacific Software Engineering Conference (APSEC '98), December 2-4, 1998, Taipei, Taiwan*. IEEE Computer Society, 1998. 3

[HHK98b] A. Hamie, J. Howse, and S. Kent. Navigation expressions in object-oriented modelling. In E. Astesiano, editor, *Proceedings Fundamental Approaches to Software Engineering, 1st Internationsl Conference, FASE'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software,*

ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, volume 1382 of *LNCS*, pages 123–?? Springer, 1998. 21

[HK99] M. Hitz and G. Kappel. *UML@Work: Von der Analyse zur Realisierung*. dpunkt-Verlag, Heidelberg, 1999. 5

[Hoa69] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969. 144

[HR00] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff. Technical Report MCS00-16, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel, September 2000. 7

[HS96] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996. 151

[HSB99] B. Henderson-Sellers and F. Barbier. Black and white diamonds. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 550–565. Springer, 1999. 41

[Int99] International Organization for Standardization. *ISO/IEC 9075-1-1999: Information Technology – Database Language – SQL Part 1: Framework (SQL / Framework)*. International Organization for Standardization (ISO), 1999. 17

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999. 5

[JCJÖ92] I. Jacobsen, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992. 1

[JSS00] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proc. International Conference on Software Engineering (ICSE), Limerick, Ireland, June 2000*, pages 730–733. 2000. 27, 134

[KW00] A. Kleppe and J. Warmer. Extending OCL to include actions. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 440–450. Springer, 2000. 3

[KWC99] A. Kleppe, J. Warmer, and S. Cook. Informal formality? the Object Constraint Language and its application in the UML metamodel. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 148–161. Springer, 1999. 21

[LB98] K. Lano and J. Bicarregui. Formalising the UML in structured temporal theories. In H. Kilov and B. Rumpe, editors, *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 105–121. Technische Universität München, TUM-I9813, 1998. 2

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreib. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, 1991. 25

[MC99] L. Mandel and M. V. Cengarle. On the expressive power of OCL. In *FM'99 - Formal Methods. World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999. Proceedings, Volume I*, volume 1708 of *LNCS*, pages 854–874. Springer, 1999. 3, 111, 112, 138

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997. 144

[Mot96] R. Motschnig-Pitrik. Analyzing the notions of attribute, aggregate, part and member in data/knowledge modeling. *The Journal of Systems and Software*, 33(2):113–122, May 1996. 41

[MS01] S. Morris and G. Spanoudakis. UML: An evaluation of the visual syntax of the language. In R. H. Sprague, Jr., editor, *Proc. 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*. IEEE Computer Society, 2001. 10

[Mut00] D. Muthiayen. *Real-Time Reactive System Development - A Formal Approach Based on UML and PVS*. Ph.D. thesis, Department of Computer Science at Concordia University, Montreal, Canada, January 2000. 19, 134

[OK99] I. Oliver and S. Kent. Validation of object-oriented models using animation. In *Proceedings of EuroMicro'99, Milan, Italy*. September 1999. 134

[Oli99] I. Oliver. 'Executing' the OCL. In A. Rashid, D. Parsons, and A. Telea, editors, *Proceedings of the ECOOP'99 Workshop for PhD Students in OO Systems (PhDOOS '99)*. 1999. 134

[OMG98] OMG. Action Semantics for the UML: Request For Proposal. OMG Document: ad/98-11-01, Object Management Group, Inc., Framingham, Mass., Internet: http://www.omg.org, 1998. 152

[OMG99a] OMG. *Meta Object Facility (MOF) Specification, Version 1.3 RTF, 2 July 1999*. Object Management Group, Inc., Framingham, Mass., Internet: http://www.omg.org, 1999. 19, 116

[OMG99b] OMG. Object Constraint Language Specification. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [OMG99c], chapter 7. 1, 6, 12, 14, 20, 24, 34, 54, 56, 59, 69, 72, 75, 76, 94, 96, 101, 102, 113, 116, 117, 121, 123, 124, 128, 135, 159

[OMG99c] OMG. *OMG Unified Modeling Language Specification, Version 1.3, June 1999*. Object Management Group, Inc., Framingham, Mass., Internet: http://www.omg.org, 1999. 1, 6, 62, 116, 197

[OMG99d] OMG. UML Notation Guide. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [OMG99c], chapter 3. 6, 8, 10, 35, 60

[OMG99e] OMG. UML Semantics. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [OMG99c], chapter 2. 6, 13, 31, 41, 50, 85, 116, 117, 119, 129, 135, 137, 147, 148

[OMG99f] OMG. *XML Metadata Interchange (XMI) Version 1.1, October 25, 1999*. Object Management Group, Inc., Framingham, Mass., Internet: http://www.omg.org, 1999. 19, 149

[OMG00a] OMG. Common Warehouse Metamodel (CWM) Specification. OMG Document: ad/2000-01-01, Object Management Group, Inc., Framingham, Mass., Internet: http://www.omg.org, 2000. 155

[OMG00b] OMG. Request For Proposal: UML 2.0 Infrastructure RFP. OMG Document: ad/2000-09-01, Object Management Group, Inc., Framingham, Mass., Internet: http://www.omg.org, 2000. 156

[OMG00c] OMG. Request For Proposal: UML 2.0 OCL RFP. OMG Document: ad/2000-09-03, Object Management Group, Inc., Framingham, Mass., Internet: http://www.omg.org, 2000. 156

[ÖP99] G. Övergaard and K. Palmkvist. A formal approach to use cases and their relationships. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 406–418. Springer, 1999. 2

[Öve00] G. Övergaard. Formal specification of object-oriented metamodelling. In T. Maibaum, editor, *Proc. Fundamental Approaches to Software Engineering (FASE 2000), Berlin, Germany*, volume 1783 of *LNCS*. Springer, 2000. 7

[PHK+99] G. Popp, F. Huber, I. Krüger, B. Rumpe, and W. Schwerin. Internet-Buchhandel – Eine Fallstudie für die Anwendung von Softwareentwicklungstechniken mit der UML. Technical Report TUM-I9915, Technische Universität München, September 1999. 19

[Pri97] S. Pribbenow. What's a part? On formalizing part-whole relations. In *Foundations of Computer Science: Potential – Theory – Cognition*, volume 1337 of *LNCS*, pages 399–406. Springer, 1997. 41

[PS89] C. Parent and S. Spaccapietra. Complex Objects Modeling: An Entity-Relationship-Approach. In S. Abiteboul, P. C. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects, Papers from the Workshop "Theory and Applications of Nested Relations and Complex Objects", Darmstadt, Germany, April 6-8, 1987*, volume 361 of *LNCS*, pages 272–296. 1989. 17, 30

[R+01] J. Robbins et al. Argo/UML CASE tool, 2001. Internet: http://www.argouml.org. 20

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs (NJ), 1991. 1, 5, 27, 30

[RG97a] M. Richters and M. Gogolla. A Web-based Animator for Object Specifications in a Persistent Environment. In M. Dauchet and M. Bidoit, editors, *TAPSOFT '97: theory and practice of software development: 7th International Joint Conference*

CAAP/FASE, Lille, France, April 14–18, 1997: proceedings, volume 1214 of *LNCS*, pages 867–870. Springer, New York, NY, USA, 1997. 14

[RG97b] M. Richters and M. Gogolla. A Web-based Animator for Validating Object Specifications. In B. C. Desai and B. Eaglestone, editors, *IDEAS'97: International Database Engineering & Applications Symposium, Montreal, Canada, August 25–27, 1997*, pages 211–219. 1997. 14

[RG98] M. Richters and M. Gogolla. On formalizing the UML Object Constraint Language OCL. In T. W. Ling, S. Ram, and M. L. Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *LNCS*, pages 449–464. Springer, 1998. 4, 49

[RG99a] M. Richters and M. Gogolla. A metamodel for OCL. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 156–171. Springer, 1999. 3, 4, 20, 115, 137, 156

[RG99b] M. Richters and M. Gogolla. On the need for a precise OCL semantics. In R. France, B. Rumpe, B. Henderson-Sellers, J.-M. Bruel, and A. Moreira, editors, *Proc. OOPSLA Workshop "Rigorous Modeling and Analysis with the UML: Challenges and Limitations"*. Colorado State University, Fort Collins, Colorado, 1999. 1, 4

[RG00a] O. Radfelder and M. Gogolla. On better understanding UML diagrams through interactive three-dimensional visualization and animation. In V. D. Gesu, S. Levialdi, and L. Tarantino, editors, *Proc. Advanced Visual Interfaces (AVI'2000)*, pages 292–295. ACM Press, New York, 2000. 10

[RG00b] M. Richters and M. Gogolla. A semantics for OCL pre- and postconditions. In T. Clark and J. Warmer, editors, *UML 2.0 - The Future of the UML Object Constraint Language (OCL), UML 2000 Workshop, York*. 2000. 4

[RG00c] M. Richters and M. Gogolla. Validating UML models and OCL constraints. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 265–277. Springer, 2000. 4, 20, 133, 144, 147

[RG00d] M. Richters and M. Gogolla. Validierung von UML-Modellen und OCL-Bedingungen. In M. Wirsing, M. Gogolla, H.-J. Kreowski, T. Nipkow, and W. Reif, editors, *Proc. GI'2000 Workshop Rigorose Entwicklung von software-intensiver Systeme*, pages 21–32. LMU München, Informatik-Bericht Nr. 0005, 2000. 4, 133

[Ric00] M. Richters. The UML Bibliography, Internet: `http://www.db.informatik.uni-bremen.de/umlbib/`, 2000. Online bibliography with references to publications about the Unified Modeling Language (UML). 2

[Ric01] M. Richters. The USE tool: A UML-based specification environment, 2001. Internet: `http://www.db.informatik.uni-bremen.de/projects/USE/`. 20, 133, 147

[RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998. 1, 5, 8, 33, 36, 85, 86

[RM99] S. Ramakrishnan and J. McGregor. Extending OCL to support temporal operators. In *Proceedings of the 21st International Conference on Software Engineering (ICSE99) Workshop on Testing Distributed Component-Based Systems, LA, May 16 - 22, 1999*. 1999. 3

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*. World Scientific, 1997. 10

[Rum98] B. Rumpe. A note on semantics (with an emphasis on UML). In H. Kilov and B. Rumpe, editors, *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 177–197. Technische Universität München, TUM-I9813, 1998. 7

[Sch97] U. Schöning. *Algorithmen – Kurz gefasst*. Spektrum Akademischer Verlag, 1997. 112

[SK98] M. Schader and A. Korthaus, editors. *The Unified Modeling Language – Technical Aspects and Applications*. Physica-Verlag, Heidelberg, 1998. 1

[SS00] S. Sendall and A. Strohmeier. From use cases to system operation specifications. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October*

2000, Proceedings, volume 1939 of *LNCS*, pages 1–15. Springer, 2000. 3

[Ste97] W. Stein. *Objektorientierte Analysemethoden*. Spektrum Akademischer Verlag, 2 edition, 1997. 6

[SW98] A. Schürr and A. Winter. Formal definition of UML's package concept. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications*, pages 144–159. Physica-Verlag, Heidelberg, 1998. 2, 7

[Tai96] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996. 30

[TE00] A. Tsiolakis and H. Ehrig. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig and G. Taentzer, editors, *Proc. of Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems, Berlin, March 2000*. 2000. Technical Report no. 2000/2, Technical University of Berlin. 10

[Tho99] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2nd edition, 1999. 92

[Ull82] J. D. Ullman. *Principles of Database Systems*. Computer Science Press, 1982. 111

[VHG+93] N. Vlachantonis, R. Herzig, M. Gogolla, G. Denker, S. Conrad, and H.-D. Ehrich. Towards reliable information systems: The KorSo approach. In C. Rolland, F. Bodart, and C. Cauvet, editors, *Advanced Information Systems Engineering, Proc. 5th CAiSE'93*, volume 685 of *LNCS*, pages 463–482. Springer, 1993. 85

[VJ99] M. Vaziri and D. Jackson. Some shortcomings of OCL, the Object Constraint Language of UML, December 1999. Response to Object Management Group's Request for Information on UML 2.0. 28

[WHCS97] J. Warmer, J. Hogg, S. Cook, and B. Selic. Experience with formal specification of CMM and UML. In H. Kilov and B. Rumpe, editors, *Proceedings ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques*, pages 167–171. Technische Universität München, TUM-I9725, 1997. 7, 21

[Whi96] R. Whitty. Object-oriented metrics: An annotated bibliography. *ACM SIGPLAN Notices*, 31(4):45–75, April 1996. 151

[Wit00] M. Wittmann. *Ein Interpreter für OCL.* Diplomarbeit, Ludwig-Maximilians-Universität München, 2000. 20

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998. 1, 12, 14, 19, 85, 123, 135

[WK99] J. Warmer and A. Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, May 1999. 1

[WT91] G. Wei and T. Teorey. The ORAC Model: A Unified View of Data Abstraction. In T. J. Teorey, editor, *Proc. 10th Int. Conf. on Entity-Relationship Approach (ER'91). ER Institute, Pittsburgh (CA)*, pages 31–58. 1991. 17, 30

List of Figures

List of Tables

Index