Technical Report 168

A Note on Design Decisions of a Formalization of the OCL
— *The View of Freiburg* —

Achim D. Brucker Burkhart Wolff

January 23, 2002

{brucker,wolff}@informatik.uni-freiburg.de
www.informatik.uni-freiburg.de/~{brucker,wolff}

Institut für Informatik
Georges-Köhler-Allee 52
79110 Freiburg, Germany

Interactive Objects Software GmbH
Basler Straße 65
79100 Freiburg, Germany

Abstract

We compare several formal and informal approaches to define the semantics of the Object Constraint Language (OCL) [22]. This comparison reveals a number of minor and major design problems to be settled in upcoming versions of the OCL standard. We review these problems in the context of our work of providing a formal semantics of OCL through an conservative embedding in HOL using the Isabelle theorem prover.[1].

Keywords: UML, OCL, formal semantics, HOL, Isabelle

[1]This work was partially funded by Interactive Objects Software GmbH (`www.io-software.com`) in a collaboration with the Software Engineering Group at the University Freiburg.

Contents

1 Introduction **7**

2 A Brief Informal Introduction to UML/OCL **9**
2.1 A Simple Example 9
2.2 Where to Use OCL in UML Diagrams 9

3 Approaches for Formalizing the OCL **11**

4 Semantic Concerns about UML/OCL **13**
4.1 On Object Models and Class Hierarchies 14
4.1.1 Dynamically Extensible Class Hierarchies 14
4.1.2 Access Modifiers: Visibilities 15
4.1.3 Finalization of Class Hierarchies 15
4.1.4 Smashing of Collections 15
4.1.5 Flattening of Collection 16
4.1.6 Navigations over Associations with Multiplicity Zero or One . . 17
4.1.7 Invariants 18
4.1.8 The role of self in the OCL standard 19
4.1.9 Object-IDs (self) and identity 19
4.1.10 The OCL type system 20
4.1.11 OCL Formulae and Inheritance: Non-monotonic Extensions . . . 21
4.2 On System States and System State Relations 21
4.2.1 The Role of @pre 21
4.3 On Operations and Methods 22
4.3.1 Strictness 22
4.3.2 Recursion 23
4.3.3 Definition of collection->sum():T 24
4.4 The OCL logic 24
4.4.1 Discussion of allInstances() 24
4.4.2 Context Declaration 25

5 Discussion: Concepts of OCL and their Interaction **27**
5.1 Discussion: OCL 1.4 vs. the Proposals for OCL 2.0 27

5.2 Summary 28
5.3 What Concepts are Expensive in HOL-OCL 28

Bibliography **31**

1 Introduction

The Object Constraint Language (OCL) [22, 33, 34] is a textual extension of the Unified Modeling Language (UML) [23] based on mathematical logic. Thus, OCL is in the tradition of other data-oriented formal specification languages like Z [1, 30, 37] or VDM [14]. For short, OCL is a side-effect free classical logic with equality and undefinedness that allows for specifying constraints on graphs of object instances.

In order to achieve a maximum of acceptance in industry, OCL is currently developed within an open standardization process by the OMG, particularly since version 1.3 the OCL is a part of the UML standard. In contrast to the various specification languages developed in the research communities, the OCL standard does not (yet) provide a formally defined semantics. Further, the syntax is only given by a grammar description; there is no metamodel, like the metamodel for the UML, given for the OCL, thus no well-formedness rules are given. The description of the OCL is merely an informal requirement analysis document with many examples, which are sometimes even contradictory. Consequently, no well-formed deduction rules are described in the standard so far.

In order to clarify the concepts of OCL formally and to put them into perspective of more standard semantic terminology, we started to provide a conservative embedding of OCL into Isabelle/HOL (As already done for other, formal specification languages; see [17] for an embedding of Z [1] into Isabelle/HOL). Thus, we attempted to built a provably consistent model for the language. As far as this was possible, we tried to follow the design decisions of OCL 1.4 in order to provide scientific insight into the possible design choices to be made in the current standardization process.

We are not describing our embedding here — this is subject of an own forthcoming paper[1]. Rather, the purpose of this document is to summarize some experiences and observations made during our work and to provide some insight into the interdependence of the impending design decisions.

This paper proceeds as follows: After a general introduction to UML and its add-on OCL by example, we will briefly discuss previous attempts to formalize OCL — including the new proposal OCL 2.0, which is an important step toward a formalization, but has no official status so far and and reveals a number of shortcomings and probably undesired effects, as we will see. The fourth chapter contains the technical core

[1] to be submitted to TPHOLs 2002

of this paper: Following the overall organization of our semantics of OCL, we present critical points of the OCL 1.4 standard, by contrasting the original informal specification with informal mathematical arguments revealing alternatives in its interpretation. In the last chapter we will summarize our tour d'horizon and our own perspective on design decisions for the semantics of OCL drawn from our experience with HOL-OCL.

2 A Brief Informal Introduction to UML/OCL

This section gives a brief introduction to UML/OCL by example; an expert reader is invited to skip it.

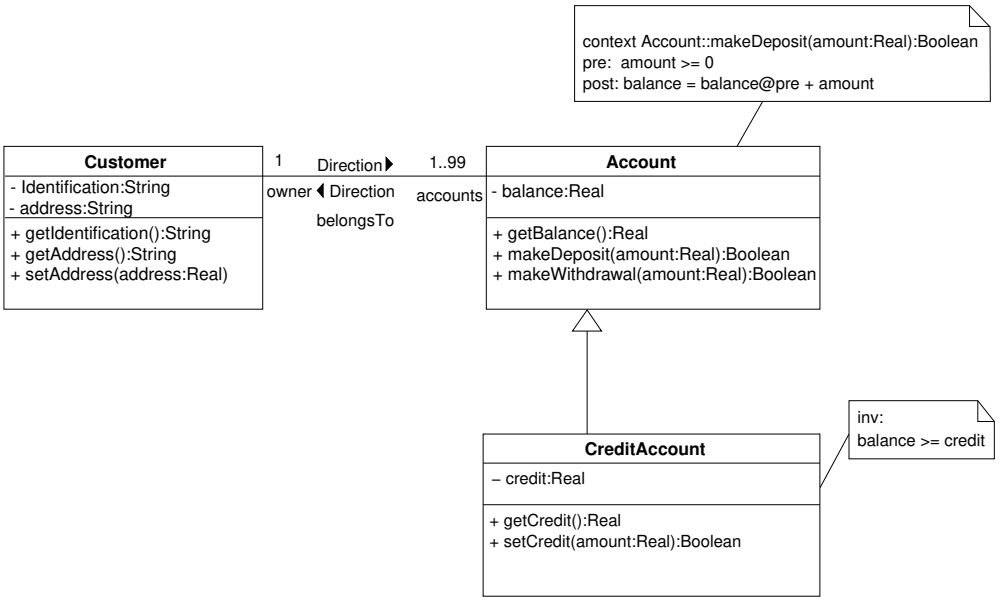
2.1 A Simple Example

In figure 2.1 an example of an UML class diagram, annotated with OCL constraint is given. The class diagram shows a simple banking scenario. The functional behavior of the methods makeDeposit(), belonging to the class **Account**, is given by the specification of its precondition and post-condition. This class has several methods and one sub-class **CreditAccount** which is inherited from **Account**. The purpose of the class **CreditAccount** is introducing a new account type in our system, which allows to make debts only up to a specific credit limit. Further we have a class **Customer** with some attributes and methods. We also model a relation between **Account** and **Customer**, in UML such a relation is called an *association*: Every instance of the class. **Account** "belongs to" an instance of class **Customer**. In detail, the association belongsTo requires, that every instance of class **Account** is associated with exactly one instance of class **Customer**. It characterizes that every account has a unique owner. In the other direction, the association models that an instance of class **Customer** is related to a set of instances of class **Account**, the size of this set is limited to number between one and 99. In this aspect, the association characterizes, that a customer must have at least one account and no more than 99 accounts. Being of type **Account** means in this context to be an instance of class **Account** or class **CreditAccount**. The range limits for association are called *multiplicities* in the UML.

2.2 Where to Use OCL in UML Diagrams

The OMG presented OCL as an universal language for expressing constraints in an object oriented context. The UML standard itself uses OCL within the "UML semantics" chapter for formally describing the well-formedness rules over UML diagrams. On the other side, within the OCL chapter, all examples presented are in the context of class

Figure 2.1 Modeling a simple bank scenario with UML



diagrams, nevertheless, the OCL standard [22] suggests the following use of OCL for enriching the UML:

Object Constraint Language Specification [22] (version 1.4), page 6–50

- invariants on classes and types in the class model,
- type invariants for stereotypes,
- specification of pre- and post conditions on operations and methods,
- guards
- navigation in data structures
- constraints on operations

We are focusing our effort on using OCL for making class diagrams more precise, and thus allowing logical reasoning over OCL formulae (and the underlying object oriented data model). Other approaches are discussed in chapter 3.

3 Approaches for Formalizing the OCL

Investigations in the research community showed several shortcomings of OCL and the need for formal syntax and semantics [32, 19, 11]. Therefore several research groups are working on formalizing OCL. These groups are focusing on different applications for OCL. Different definitions of the OCL syntax using a metamodel (in analogy to the UML standard) are given in [4, 28, 3].

Based on a intuitive operational semantics the university Dresden developed an OCL type checker and Java based constraint checking code generator was implemented [9, 35].

Recently a formal operational semantics together with a formal type system for OCL 1.4 was presented in [20]. The authors focus on the issue of subject reduction — i.e. a compatibility property of the type system with the operational semantics — but do not define the semantic function for expressions whose evaluation diverges.

The most recognized approach for a formal semantics (at least from the perspective of the OMG) has been carried out by the database group at the university Bremen [27, 28, 29]. The Bremen approach is much inspired by algebraic specification techniques and is based on a preconceived, but generic static class hierarchy (see discussion about "universes" in Section 4.1.1).

Both approaches have in common that they capture only subsets of the standard (selected with respect to specific application areas such as data-base query languages within OCL etc.). Moreover, they are based on "mathematical notation" in the style of "naive set theory", which is in our view quite inadequate to cover such evolved subjects such as inheritance and a considerable step back from other standardization efforts such as Z [1], where Zermelo-Fränkel set-theory is explicitly used and worked out with a considerable degree of detail. Moreover, both approaches do not attempt to provide a formalization precise enough and suited for mechanization in a theorem prover.

The proposal OCL 2.0 represents a considerable step toward a formalization. It evolved from the Bremen approach to OCL semantics, but has no official status so far and reveals in our view a number of shortcomings and probably undesired effects. (see section "Discussion" at the end of this paper).

4 Semantic Concerns about UML/OCL

The semantics of UML/OCL has to be built in several layers:

- the *object model* layer, that provides the concepts of a class-hierarchy, typed constructors and destructors of data structures,
- the *system state* containing essentially a graphs of data constructed along a class hierarchy,
- *relations* on system states, (forming the denotational domain for the semantics of methods),
- *the OCL logic* consisting of operators that enable us to describe relations on states (for describing methods in a pre-/post-condition style).

OCL is based on UML class diagrams[1]. Roughly speaking, object models consist of (sets of) trees, where each node represents a "class" and each edge the inheritance relation. For simplicity, it is possible to relate all roots of the trees in this set to a special superclass OclAny; thus, without loss of generality, we can consider the class hierarchy (with an special exception made for the OCL collection types) as one single tree with OclAny as root.

Classes in a hierarchy provide two different entities: *attributes* and *methods*. The attributes of a class can be seen as components in a record, an *object*. The name of an attribute can be used to select the components in an object; sequences of such selections form so-called *path expressions* allowing the navigation in object structures[2].

Objects constructed according to a class hierarchy are linked via implicit references or *unique object identifiers* in a (possibly cyclic) graph, the *system state*. These graphs are meant when talking over "associations" and their "cardinalities". Since references in objects may be undefined within an object model of OCL, the higher constructs of the language must cope with possibly undefined (path-) expressions. Other erroneous evaluations such as divide by zero where handled by undefinedness as well.

In principle, methods of a class are modeled as relations on system states. Since paths may be constructed such that they refer both on the object model before and after

[1] Attempts to use OCL within other diagram types are far too premature to be discussed within a formal semantics.

[2] However, OCL provides no syntax for constructors of objects

a method call, it is necessary to build the operators of the OCL logic as interpretations on object model relations.

In the following, we will describe these four layers in more detail and use the layering to structure our critique on (the various versions of) the OCL language design.

4.1 On Object Models and Class Hierarchies

4.1.1 Dynamically Extensible Class Hierarchies

Within the standard UML/OCL, it is self-understood that class diagrams are extensible — in a given class hierarchy (see figure 2.1) **A**, a new group of classes may be added at any possible leaf or node of **A**. In our example (see figure 2.1) one could add a new type of credit accounts by inheriting from **CreditAccount** or add a class **Clerk** without a direct link to a currently existing class. Within a group, added classes may have mutual static dependencies between types and method calls. Dependency cycles may arise as a consequence of extensions; for a given hierarchy, it is hence not predictable that some part of a class diagram will not depend on another after some extension. Nevertheless, it is highly desirable that proofs done for one class hierarchy also hold for its extensions.

The data (the "objects") generated via *constructors* of classes may also have mutual dependencies — actually, it is hard to avoid the idea of general graphs, although this is stated nowhere in the standard. Since data dependencies can be also cyclic, at least co-inductive datatype constructions would be necessary for its semantic representation. However, since the nature of these cycles is only clear within a given class hierarchy (hence the number of elements in a co-inductive data-type statement is not fixed), the semantics for these is a further challenge for proof support by co-inductive datatype packages [26].

Living with a dynamic data model represents a major challenge for a formal semantics of OCL — especially if the subtypes resulting from class hierarchy extensions have to "fit" into methods defined earlier on supertypes on them. If one chooses a typed logical meta-language, than subtyping of OCL must be embedded into the type notion of this meta-language.

Semantically, it is considerable to construct as domain of the OCL-semantics a "super universe" that comprises all carrier sets for types induced by all class hierarchies closed under extension. Unfortunately, such a universe is too large (in a set-theoretic sense) to be representable inside the standard models of higher-order logic (HOL; for models in the 'standard'-sense, cf. [2, 10]), at least as long as we represent methods as total functions. A theoretically viable alternative would be to represent methods as "continuous functions" and to embed them in a so called "reflexive domain" of D. Scott's D_∞-model (see [36] for a more recent account). Such an approach would require a (conservative) Isabelle-theory for reflexive domains, which is not yet available. Moreover,

this approach would internalize all typing into set-reasoning inside the logic (similarly to Isabelle's ZF-theory), which is pragmatically not desirable since we are more interested in an embedding that is amenable to applications instead of meta-theory.

For this reasons, we avoid a "super universe"-approach; instead, in our semantic embedding, we exploit the fact that any concrete class-hierarchy is finite and provide semantic mechanisms to cope with dynamic extensions.

Both semi-formal semantic definitions do not cope with the aspect of dynamism; they declare OCL semantics only in terms of a given, pre-conceived class hierarchy. They essentially construct a new "denotational domain" for each new class hierarchy. Since a new domain is not related to an old one other than by a pair of surjective-injective morphisms, it is technically non-trivial to save proofs built upon a former class hierarchy. Any semantic representation of OCL usable for a theorem prover will have to find an answer to this problem.

4.1.2 Access Modifiers: Visibilities

The access modifiers private, protected or public in UML are called *visibility*. They have the usual semantics (for example, like in Java [15, section "6.6 Access Control"]). It has to be discussed, what kind of (static) visibility rules hold for constraints annotating methods or classes. As an alternative, one could interpret the access to private variables in OCL-formulas as *undefined* $\bot$, but this results in a quite complex calculus and problems with respect to the executability of (a fragment of) the OCL.

4.1.3 Finalization of Class Hierarchies

Many of the well known object oriented programming languages, like Java, provide a special keyword for marking classes that are not extensible, e.g. in Java this statement is called finalize.

In contrast to this tradition, the UML standard seems not to introduce a statement for finalization, nevertheless our formalization will be able to handle finalization, which can be useful when doing reasoning over class hierarchies.

4.1.4 Smashing of Collections

For Collections (Sets, Bags, Sequences, and Tuples[3]), the question arises how to handle the case of undefined values inserted into a collection. There are essentially two different possibilities for their treatment: Tuples, for instance, may be defined:

$$(\bot, X) = (Y, \bot) = \bot$$

with the consequence:

[3] only in [34]

first (X, $\perp$) = $\perp$ second($\perp$, Y) = $\perp$

or, in contrast:

($\perp$, X) $\neq$ (Y, $\perp$) $\neq \perp$

with the natural consequence:

first (X, $\perp$) = X second($\perp$, Y) = Y

In the literature [36, 21], the former version is called a "smashed product", while the latter is just the standard product. We also apply this terminology for sets, bags and sequences. Smashing of Collections (or not) is directly connected to the question whether the constructors of collections are strict (or not):

{$\perp$, X} = {Y, $\perp$} = $\perp$
<1, 2, $\perp$> = $\perp$

et cetera.

4.1.5 Flattening of Collection

The OCL standard postulates that Collection of Collection are automatically flattened, therefore hierarchical set constructs are not available within OCL (but could be constructed within user defined UML/OCL data types). Astonishingly, the flattening process is only defined by the following (trivial) example:

Object Constraint Language Specification [22] (version 1.4), page 6–67

> Within OCL, all Collections of Collections are flattened automatically therefore, the following two expressions have the same value:
>
> **Set**{**Set** {1, 2}, **Set** {3, 4}, **Set** {5, 6}}
> **Set** {1, 2, 3, 4, 5, 6}

This is one of the most controversial discussed points of the OCL standard. Among other problems, it is unclear, how nested collections such as sets-of-lists or bags-of-sequences-of-sets should be treated. As a consequence, recent approaches for formalizing the OCL avoid flattening. For example, the semantics presented in [27] is based on non-flattened sets.

In the OCL 2.0 proposal [34, table 5.7] a particular flattening strategy, which we call *flattening to the first*, is proposed. The central concept, is that the first collection type "wins", e.g. Sequence(Set(A)) is flattened to Sequence(A). Unfortunately sequences have a stronger algebraic structure than sets. As a result, there are several possibilities to define the flattening operation in these cases, e.g.

Sequence{**Set** {3, 4}, **Set**{1, 2}}

Table 4.1 Flattening of nested collections

Nested collection types	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))	Set(t)
Bag(Bag(t))	Bag(t)
Sequence(Sequence(t))	Sequence(t)
Sequence(Set(t))	Set(t)
Sequence(Bag(t))	Bag(t)

can be flattened to

1. Sequence 1, 2, 3, 4
2. Sequence 3, 2, 1, 4
3. Sequence 4, 3, 2, 1
4. ... (up to 4! possibilities)

Thus [34] is semantically under-defined. There a two ways out of this:

1. The OCL standard could leave to an tool implementor, what particular choice is made (in the past, for languages like C, this preceding proofed to be the worst).
2. One could explicitly define an ordering between all objects of OCL. This approach is feasible in principle, but quite complex in details.

There is an alternative to flattening to first, that we call *flattening to the weakest* (i.e. Sequence(Set(A)) is Set(A) and not Sequence(A)). In this view, sequences, bags, and sets are ordered by their algebraic strength regarding their structure (monoids are stronger than monoids with commutativity are stronger than semi-lattices). Flattening to the weakest (see table 4.1) semantically possible without requiring an ordering.

4.1.6 Navigations over Associations with Multiplicity Zero or One

For navigations over associations with multiplicity zero or one, the UML standard states a equivocally semantics:

Object Constraint Language Specification [22] (version 1.4), page 6–60

> Because the multiplicity of the role manager is one, self.manager is an object of type Person. Such a single object can be used as a Set as well. It then behaves as if it is a Set containing the single object.

Within OCL the choice of the "accessor-operator" (. or ->) decides, if the attribute is set or a singular type, this interpretation seems to be syntactically consistent. Nevertheless this type dualism does not harmonized with a stricter type system as it will be needed for our formalization.

At current, we will only allow a set based type for association ends within our embedding, but we will consider a conversion of external specifications during parsing.

4.1.7 Invariants

The UML standard introduces special OCL formulae: invariants, pre- and post-conditions. Pre- and post-conditions are specific to a method and should hold directly before, respectively directly after, the method invocation. For invariants, there is a the following postulation (which also is written in the OCL 2.0 RfP [34]):

Object Constraint Language Specification [22] (version 1.4), page 6–52

> An OCL expression is an invariant of the type and must be true for all instances of that type at any time.

As explained in [8] we will relax this postulation by introducing different types of invariants:

Class invariants: We denote invariants attached to classes or attributes as *class invariants* and postulate that they must be true for all instances of that class at any time.

Method invariants: We call invariants attached to methods as *method invariants* and postulate that they must hold before and after the method invocation. Method invariants are a shorthand for formulae that should are part of the pre- and post-condition of a method. Particularly, we allow, that method invariants can be violated during the method invocation.

Association invariants: We denote invariants belonging to associations as *associations invariants*, which we convert to class invariants at all opposite ends (see [7, 22] for details). Note, that multiplicities are syntactical notation for association invariants, see [7] for details.

Invariants on other classifiers: At the moment, we restrict ourself to invariants used in the context of class diagrams.

Summarizing, we propose to distinguish class invariants (that correspond to invariants on data-types and would be handled in Z, for instance, in a schema representing a data entity) from method invariants, that simply abbreviate formulae occurring both in pre-conditions *and* post-conditions. This means that method invariants only hold when entering and leaving a method — which leaves enough flexibility for a powerful Hoare-like calculus — while class invariants really hold "at any time".

This denotational semantics is stricter than the operational semantics presented in [8, 7], in particular the operational semantics only checks *all* kinds of invariants at the entry and exit points of method calls.

Possibly, we have to introduce several other variation of invariants, for example *component invariants* as described in [12].

4.1.8 The role of self in the OCL standard

In OCL standard introduces a reserved word self. At a moment, this could be associated with a kind of an unique object identifier, but the standard states only:

> Object Constraint Language Specification [22] (version 1.4), page 6–51
>
> Each OCL expression is written in the context of an instance of a specific type. In an OCL expression, the reserved word self is used to refer to the contextual instance.

That means, that self is in no way special. In our semantics it is only a all quantified variable of the corresponding type.

In general, self can be replaced by a variable of the type of the classifier described in the context. This formalization conforms also to the suggestions for OCL 2.0 [34].

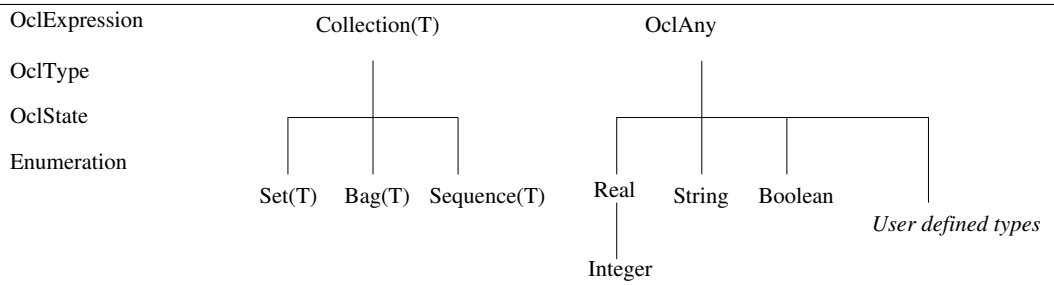
4.1.9 Object-IDs (self) and identity

Even if the standard does not propose a unique identifier (object reference) for every object, as usually available in object oriented programming languages (often called self, as in Java, or this, as in C++), our semantic representation of data is designed to be easily extensible by such a construct. But introducing a unique object identifier would imply a great hassle, particularly several variants of equality operations would be needed, at least:

$a \doteq b$**:** Returns true, if and only if a and b are from the same type and all attributes are equal (*equality on values*).

$a \doteqdot b$**:** Returns true, if and only if a and b have the same (unique) object identity (*equality on objects*).

Figure 4.1 The type system of OCL



This distinct equality operation will lead to several new operations around it, for example we would have to answer the questions: *What is the semantic of the* includes *relation?* In detail we have to decide, if we define includes in relation to "$\doteq$" or in relation to "$\doteqdot$".

Note that undefinedness $\perp$ makes already several different equalities (strict *weak* equality vs. *strong* equality) necessary.

Note moreover, that the relation $(a \doteqdot b) \implies (a \doteq b)$ holds, which is easy to see.

4.1.10 The OCL type system

OCL introduces its datatypes within the UML package UML_OCL, which in principle consists of sets, numbers, strings and boolean types (see figure 4.1 for details). With regard to the standard, some points were not quite clear, therefore we lay down:

- OclState, OclExpression, OclType, Enumeration and the set hierarchy (Collection and subtypes) are not inherited from OclAny.

- The basic data types (Integer, Real, String) are (conceptually) inherited from OclAny. Even if one respects this (purely conceptually motivated, but artificial) inheritance, there are two ways to values of data-types: either in the system state or not. Having them inside the state means that implicit references can be created for them, which allows to handle them uniformly with all other data; such an approach makes the state infinite. Having values of basic data outside the store means, however, that accessing them has to be done differently as other objects. (We chose this variant).

- We will not explicitly model the OclExpression type, as it is not really needed. This conforms to the Proposal for OCL 2.0 [34].

- We will not explicitly model the OclType type, as it is not really needed. This conforms to the Proposal for OCL 2.0 [34].

Table 4.2 Refinement relations within a monotonic extension (B is inherited from A)

constraint	refinement	note
pre-condition	$A \implies B$	
post-condition	$B \implies A$	
method-invariant	$A \iff B$	
class-invariant	$B \implies B$	

4.1.11 OCL Formulae and Inheritance: Non-monotonic Extensions

The OCl standard doesn't gives a hint, how OCL specification are "inherited". One idea of handling this implemented by Eiffel's subcontracting: *Preconditions may only be weakened while post-conditions may only be strengthened.* An additional postulation, that invariants may also only be strengthened, or more formal, the refinement relations shown in table 4.2 are holding.

This allows only monotonic extensions, which means, that all clauses holding before an extension also hold in the extension. Further, this guarantees the compliance with the Liskov principle [18]. With a loss of generality, we postulate that only type hierarchies respecting the Liskov principle, including its OCL formulae (and thus our refinement) are well typed.

4.2 On System States and System State Relations

4.2.1 The Role of @pre

Within the OCL, the postfix operation @pre plays an important role in introducing the state concept. It allows within accessing of the pre-state within a post-condition formula.

Whereas our semantics forbids the use of the postfix operation @pre for the basic data types (Integer, Real, String and Boolean), the OCL standard does not limit its use. Given the class Account (see figure 2.1) with the attribute balance, which is a basic data type, we define, that Account@pre.balance and Account.balance@pre have the same semantics. Therefore this is no loss of generality (from this viewpoint, its only syntax). *But in our semantics*, there is no way to write Account@pre.balance@pre, which could is syntacticly allowed in the standard (see page 6–68 for an example of using @pre operator with an basic data type as argument). Further @pre can also used as a postfix operator on method calls, returning the result if the function was called in the previous state.

Further, the OCL standard gives only vague idea of the semantic of path expression containing several several @pre operations, e.g. Customer@pre.accounts@pre.balance:

Object Constraint Language Specification [22] (version 1.4), page 6–68

When the pre-value of a property evaluates to an object, all further properties that are accessed of this object are the new values (upon completion of the operation) of this object. So:

a.b**@pre**.c —— *takes the old value of property b of a,*
—— *say x and then the new value of c of x*
a.b**@pre**.c**@pre** —— *takes the old value of property b of a,*
—— *say x and then the old value of c of x*

In [13] a detailed discussion of this problematic from the view point of an tool vendor (based on an operational semantics) is given. For our formalization there is no need to restrict the use of @pre in this way.

4.3 On Operations and Methods

4.3.1 Strictness

Since path expressions may be undefined in a concrete state, all types are implicitly assumed to contain an element representing undefinedness (we call it $\perp$). The question is what arbitrary operations and methods should behave, when applied to an undefined expression. The OCL standard quite clearly favors what is called "strict extension" in the literature, with an exception made for Boolean operations [22]:

Object Constraint Language Specification [22] (version 1.4), page 6–58

Whenever an OCL expression is being evaluated, there is a possibility that one or more of the queries in the expression is undefined. If this is the case, then the complete expression will be undefined.
There are two exceptions to this for the Boolean operators:

- True OR-ed with anything is True
- False AND-ed with anything is False

The above two rules are valid irrespective of the order of the arguments and the above rules are valid whether or not the value of the other subexpression is known.

This definitions leads to a Kleene logic [16] for the Boolean values. The nice thing about Kleene logic is, that the known algebraic structure of the boolean algebra is preserved, particularly a Kleene logic is a *distributive lattice*. For example the associativity and commutativity holds for the usual boolean operations and the idempotency holds with the exception of the implication, which only works for defined values.

The main drawback of a three-valued Kleene logic is, that the axiom of the excluded middle ($a \vee \neg a = \text{true}$, mind $\bot \vee \neg \bot = \bot$) and the idempotency of the implies ($x \implies x = \text{true}$, mind $\bot \implies \bot = \bot$) operation does not hold. This has the consequence that many proof procedures for existing theorem proving systems like *grind* in PVS [24] or *fast_tac* in Isabelle [25] do not work any longer (or have to be adapted in a non-trivial way).

All other operations (including user-defined methods) should be strict in their arguments. With respect to collections, this raises the question about terms like $\{a, \bot\}$ and *smashing* which is discussed in section 4.1.4.

4.3.2 Recursion

Methods can be specified by recursion (see page 6–59), for example given a class **number** that implements a fac() the following is a correct OCL definition:

```
context number.fac(n: Integer): Integer
post:   result = if (n=0)
                 then 1
                 else n * self.fac(n−1)
                 endif
```

There are two fundamentally different approaches to interpret recursive constraints in OCL, both of which are consistent with the standard:

- The *predicative approach*. In this view, we admit all possible function interpretations for fac — that is, not just the *least* fixpoint, but *any* fixpoint that solves a recursive constraint. Surprisingly, the solution for fac(-1), for example, is just -1, which involves a small proof using induction and a lemma on dividability.
- The *denotational approach*. In this view, recursive constraints are interpreted as *least fixpoint*, defining fac(-1) as $\bot$. Semantically, this means that we have to add to any data domain including Bool a cpo[4] — structure (with an ordering, completeness properties, etc.; see [36]), saying that all data-domains are *flat cpo's* etc. This is perfectly feasible — also in Isabelle — but implies a more involved semantic foundation of OCL.

The OCL standard says the following:

> Object Constraint Language Specification [22] (version 1.4), page 6–59
>
> The right-hand-side of this definition may refer to the operation being defined (i.e. the definition may be recursive) as long as the recursion is not infinite.

[4] complete partial ordering; see [36]

This leaves open what the exact nature of statements like:

```
context number.M(n: Integer): Integer
post:    result = self.M(n)
```

should be: is it *illegal OCL*? Or is M the undefined function? If it is illegal, then it would be advisable to provide a termination ordering and to provide proofs that this ordering is respected by the arguments of all recursive calls (this is the well-foundedness-condition in systems like PVS or Isabelle). Leaving the question open is highly unsatisfactory in our view, since a check for well-foundedness is undecidable, and well-foundedness requires a non-trivial proof theory that has to be explicit to the user. Much easier for the user of OCL would be the denotational approach, where no termination ordering and termination proofs must be provided and the result for such statements is just what one gets operationally.

4.3.3 Definition of collection->sum():T

What is a well-formed definition of the sum of a arbitrary set. The OCL standards (under the assumption of finite sets) defines collection->sum():T as:

> Object Constraint Language Specification [22] (version 1.4), page 6–85
>
> The addition of all elements in a *collection*. The elements must be of a type, supporting the + Operation. The + operation must take one parameter of type T and be both associative: $(a + b) + c = a + (b + c)$ and commutative: $a + b = b + a$. Integer and Real fulfill this condition.
>
> ```
> post: result = collection->iterate(elem; acc:T = 0 |
>                                    acc + elem)
> ```

Analyzing this definition, we have to remark, that: OCL 1.4 is not strong enough for formulating the associativity and commutativity of an arbitrary $+$ operation (e.g. an precondition of sum). This is caused by the lack of an general universal quantifier (e.g. for all whole numbers should hold, ...). In contraction, our formalization will be able to express such Properties. Going even further our embedding will claim a proof for the associativity and the commutativity of the $+$ operation.

4.4 The OCL logic

4.4.1 Discussion of allInstances()

The use of this operator is discouraged in the actual OCL standard:

Object Constraint Language Specification [22] (version 1.4), page 6–66

> The use of allInstances() has some problems and its use is discouraged in most cases. The first problem is best explained by looking at the types like Integer, Real and String. For these types the meaning of allInstances() is undefined. What does it mean for an Integer to exist? The evaluation of the expression Integer.allInstances results in a infinite set and is therefore undefined within OCL. The second problem with allInstances is, that the existence of objects must be considered within some overall context, like a system or a model. This overall context must be defined, which is not done within OCL. A recommended style is to model the overall contextual system explicitly as an object within the system and navigate from that object to its containing instances without using allInstances.

This statement obviously results from the tradition to view OCL as an assertion language[5], in which assertions in themselves must be executable. From a logical point of view (having typed, infinite sets), there is no reason to exclude allInstancesOf(Integer) from being the — infinite — set of integers. This just results in a universal quantification. In our view, we would prefer to distinguish OCL as specification language from an "executable subset" of OCL, that can be used for generated checks of pre- and post-conditions, for instance. However, in our actual proposal for OCL we followed the quite explicit requirement of the standard to define allInstances to be $\perp$ (having in mind that a change here is a minor issue from our perspective).

4.4.2 Context Declaration

In the OCL standard the context seems to have a completely syntacticly meaning, it is even stated, that

Object Constraint Language Specification [22] (version 1.4), page 6–52

> The context declaration is optional.

We interpret this statement in the following way: A explicit (textual) context declaration is optional, if the context of the OCL expression is given by the UML/OCL specification. For example, a graphical notation for defining the context can be used. For example, in figure 2.1, the OCL formulae are written in "UML notes" directly linked to classes defining their context.

Further, we are giving the context statement the semantics of an universal quantification: "For all classifiers described by the context, the following OCL formulae should

[5] . . . and is in this respect somewhat similar to some initiatives to standardize hardware verification languages such as `http://www.verificationlib.org/`

hold". This interpretation of the context statements corresponds to the OCL 2.0 RfP [34] and further allows the specification of nested contexts, e.g.:

```
context c:Customer, a:Account
  inv : accounts.forall (a.owner = owner)
```

5 Discussion: Concepts of OCL and their Interaction

5.1 Discussion: OCL 1.4 vs. the Proposals for OCL 2.0

Comparing the standard OCL 1.4 with the several approaches to formalize it, two fundamental design questions of OCL have to be answered:

- What is the role of "undefinedness" in OCL: Just a symbol for exceptional behavior (like dereferencing an object reference in a state where the object does exist) or the "undefinedness" in the sense of denotational semantics: "I don't know yet the value"?

- Should OCL be operational or at least contain easy identifiable language subsets that can be compiled to code? Or should OCL be a powerful logical language like HOL?

.

While OCL 1.4 had the explicit intention to keep the language strict (in order to keep its semantics close to standard programming languages such as Java or SML in order to simplify code generation), the version 2.0 seems to drift toward standard products, sets, bags etc., that are difficult to execute (if at all), while inconsequently insisting on strictness on all other places. The overall idea in OCL 2.0 is that $\bot$ is a kind of exception, that can be tested. However, in OCL 1.4 the idea was "no information", which is compatible with the usual denotational semantics for recursive methods, that explains recursive functions semantically by limits to approximations in *complete partial orderings* called "least fixed points".

In the "no information view", the equality on elements is no longer decidable in general, and non-strict operations like count in the sense of version 2.0, for example:

are no longer executable. This is different for a smashed version of Collections, where count would have the property:

$\mathrm{count}(\{\bot\}, \bot) = \bot$

A definition of count in the latter sense is executable[1]. Recursive functions uniquely built from strict operations can be syntactically identified and are thus a simple basis for an executable subset close to a strict programming language. Of course, the exceptional behavior view for undefinedness (as imposed in OCL 2.0) is in itself also executable; however, it is neither compatible with denotational recursion semantics, nor with existing tools such as the Dresden Code Generator. Since the authors of OCL 2.0 still maintain the claim that OCL is a specification language, not a programming language[2] the overall rationale behind OCL 2.0 seems to be remarkably unclear with this respect since their proposal inherits many limitations that make only sense in connection with executability (such as finiteness of sets, etc.).

In our view the original design of OCL 1.4 (partially inspired by specification languages such as SPECTRUM [5, 6] and thus in the tradition of LCF based on denotational semantics) seemed to have a different approach toward executability, both with respect to proliferation to engineers (that are used to strict programming languages) as well as tool support of OCL. Instead of this "programming approach to specification", there is in OCL 2.0 a somewhat inconsequential step toward a universal logical language in the sense that usual quantifiers are still avoided, restricted to weird finiteness , etc. Such an in-between design of the language is prone to a two-fold failure: Failure to be usable in tools for software engineers and failing to be a success on the market of specification languages.

5.2 Summary

Clearly, 2.0 is a substantial step forward to a formalization of the OCL. However, there is also a conceptual shift with respect to strictness and the interpretation of $\bot$. Some elements were added (like tuples) that we consider an achievement, while others (finiteness of sets, undefinedness) are considered harmfully. Although a direct comparison might look a bit unfair since 2.0 contains a relatively detailed (mathematically rigorous) model while 1.4 is merely a collection of vaguely described requirements, such a comparison may be helpful to exemplify the conceptual shift and to substantiate our claim, that essential parts are still missing. For a summary, see table 5.1

5.3 What Concepts are Expensive in HOL-OCL

As already mentioned, this paper arose from a concrete formalization project of OCL using the generic theorem prover Isabelle by building a shallow, conservative embed-

[1] since it is *continuous* in the sense of complete partial orderings; note that the Kleene-Logic operators are also continuous, which was the original motivation for their definition ...

[2] although we do not understand why; apparently the authors simply say that they don't care for executability

Table 5.1 Comparison of important properties

	OCL 1.4	OCL 2.0 RfP	Freiburgs advice
dynamic universe	□	□	☑
recursion	☑	□	☑
smashing	?	□	☑
flattening	☑	□	□
tuples	□	□	☑
finite state	☑	☑	□
object id's	□	implicit	?
strict Ops	☑	□	☑
general Quantifiers	□	☑	?
allInst finite	☑	☑	□
several @pre	☑	☑	☑

☑: supported □: unsupported

Table 5.2 Costs of the several design decisions

decision	costs	note
smashing/standard	★★★	smashing slightly expensive
flattening	★★★★	
tuples	★	generally recommended
infinite state	★	
object id's	★★	possible, if needed
general quantifiers	★	recommendable for a verification calculus
allInstances infinite	★	recommendable for a verification calculus
several @pre	★	but no for basic datatypes

★: inexpensive ★★★★: expensive

ding into Higher-Order Logic (HOL). In our approach, the different semantics could be achieved by usually simply changing a line here or to re-adjust a screw there. Having this enormous flexibility, we could adopt to many different design choices easily. Still, from the perspective of our experience, we would like to give an estimation of the "costs" of various design decisions. With costs, we mean either difficulty in implementation or theorem proving complexity (see table 5.2).

Strictness in general and smashing is easy to define theoretically and also easy to implement in code generation tools. In contrast, strictness and smashing are expensively during theorem proving, since fundamental operations, such as substituting arguments into the body of an operation, pervasively requires reasoning over the definedness of these arguments. This unpleasant problem is well-known and can be managed (for example as in LCF or HOLCF), if necessary.

Flattening is perfectly feasible, but requires a number of non-standard techniques in type-checking and will turn out to be difficult to grasp by engineers at the very end, in particular in its problems with set/bag/list-combinations.

Tuples are a simple, but very useful, thing that should simply be incorporated into the standard.

Infinite states, infinite sets (as allInstances(Int)) and unbounded quantifiers are easy to implement from our perspective (they were mapped to suitable quantifiers or sets in HOL) and necessary elements for a practically useful *calculus* for OCL. They do not hamper the identification of executable subsets of OCL.

Object id's can be added to our semantics remarkably easily. However, the practical consequences for OCL as a language may be quite severe, since this implies further equalities. Besides weak equality and strong equality, we will have referential equalities and value equalities and all sorts of combinations.

Bibliography

[1] Formal Specification – Z Notation – Syntax, Type and Semantics. October 2001. http://www.cs.york.ac.uk/~ian/zstan/. Consensus Working Draft 2.7.

[2] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truh Through Proof*. Academic Press, 1986. http://gtps.math.cmu.edu/andrews.html.

[3] Thomas Baar and Reiner Hähnle. An integrated metamodel for OCL types. In R. France, editor, *OOPSLA 2000 Workshop Refactoring the UML: In Search of the Core, Minneapolis, Minnesota, USA*. October 2000. http://i12www.ira.uka.de/~key/doc/2000/baarhaehnle00.ps.gz.

[4] Marc Bodenmüller. The OCL Metamodel and the UML_OCL package. In *Workshop on the Object Constraint Language*. Canterbury, March 2000. http://www.dcs.kcl.ac.uk/staff/tony/OCL2000/BodenMueller.pdf.

[5] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, May 1993. http://www4.informatik.tu-muenchen.de/proj/korso/papers/v10.html.

[6] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part II. Technical Report TUM-I9312, Technische Universität München. Institut für Informatik, May 1993. http://www4.informatik.tu-muenchen.de/proj/korso/papers/v10.html.

[7] Achim D. Brucker and Burkhart Wolff. Checking OCL Constraints in Distributed Systems Using J2EE/EJB. Technical Report 157, Albert-Ludwigs-Universität Freiburg, July 2001. http://wailoa.informatik.uni-freiburg.de/cgi/publications/extract_abstract.cgi?KEY=brucker.ea:checking:2001.

[8] Achim D. Brucker and Burkhart Wolff. Testing distributed component bases systems using UML/OCL. In K. Bauknecht, W. Brauer, and Th. Mück, editors, *Informatik 2001*, volume 1 of *Tagungsband der GI/ÖCG Jahrestagung*, pages 608–614. Wien, November 2001. ISBN 3-85403-157-2. `http://wailoa.informatik.uni-freiburg.de/cgi/publications/extract_abstract.cgi?KEY=brucker.ea:testing:2001`.

[9] Frank Finger. *Design and Implementation of a Modular OCL Compiler*. Diploma thesis, Technische Universität Dresden, March 2000. `http://www-st.inf.tu-dresden.de/ocl/ff3/diplom.pdf`.

[10] Mike J. C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[11] A. Hamie, F. Civello, J. Howse, S. Kent, and M. Mitchell. Reflections on the Object Constraint Language. In *Post Workshop Proceedings of UML98*. Springer-Verlag, Heidelberg, June 1998. `http://www.cs.ukc.ac.uk/pubs/1998/788`.

[12] Rolf Hennicker, Hubert Baumeister, Alexander Knapp, and Martin Wirsing. Specifiying component invariants with ocl. In K. Bauknecht, W. Brauer, and Th. Mück, editors, *Informatik 2001*, volume 1 of *Tagungsband der GI/ÖCG Jahrestagung*, pages 608–614. Wien, November 2001. ISBN 3-85403-157-2.

[13] Heinrich Hussmann, Frank Finger, and Ralf Wiebicke. Using previous values in OCL postconditions: An implementation prespective. In *UML 2.0 - The Future of the UML Constraint Langauge OCL*. York, UK, October 2000. `http://www.inf.tu-dresden.de/TU/Informatik/ST2/ST/papers/uml2000_oclws.pdf`.

[14] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. `ftp://ftp.ncl.ac.uk/pub/users/ncbj/ssdvdm.ps.gz`. 0-13-880733-7.

[15] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The JavaLanguage Specification*. Addison-Wesley Europe, Amsterdam, The Netherlands, second edition, 2000. ISBN 0-201-31008-2. `http://java.sun.com/docs/books/jls/index.html`.

[16] S. C. Kleene. *Introduction to Meta Mathematics*. Wolters-Noordhoff Publishing, Amsterdam, 1971. ISBN 0-7204-2103-9. Originally published by Van Nostrand, 1952.

[17] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of z in isabelle/hol. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics — 9th International Conference*, LNCS 1125, pages 283–298. Springer Verlag, 1996.

[18] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994. ISSN 0164-0925. `http://www.acm.org/pubs/citations/journals/toplas/1994-16-6/p1811-liskov/`.

[19] Luis Mandel and Marìa Victoria Cengarle. On the expressive power of ocl. *World Congress on Formal Methods (FM'99, Proceedings)*,, September 1999. `http://www.pst.informatik.uni-muenchen.de/personen/cengarle/ocl.ps`.

[20] Luis Mandel and Marìa Victoria Cengarle. A formal semantics for ocl 1.4. In C. Kobryn M. Gogolla, editor, *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *LNCS*, pages 91–. Springer, Torontoo, Canada, October 2001. `http://link.springer.de/link/service/series/0558/tocs/t2185.htm`.

[21] P. D. Mosses. *Denotational Semantics*, chapter 11. In van Leeuwen [31], first edition, 1990.

[22] *Object Constraint Language Specification*, chapter 6. In Object Management Group [23], February 2001. `ftp://ftp.omg.org/pub/docs/ad/01-02-13.pdfl`. Version 1.4.

[23] OMG Unified Modeling Language Specification (draft). February 2001. `ftp://ftp.omg.org/pub/docs/ad/01-02-13.pdfl`. Version 1.4.

[24] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, New Brunswick, NJ, July/August 1996. `http://pvs.csl.sri.com`.

[25] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828. Springer-Verlag Inc., New York, NY, USA, 1994. ISBN 3-540-58244-4, 0-387-58244-4, xvii + 321 pages.

[26] Lawrence C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 187–211. MIT Press, 2000. `http://www.cl.cam.ac.uk/users/lcp/papers/Sets/milner-ind-defs.pdf`.

[27] Mark Richters and Martin Gogolla. On Formalizing the UML Object Constraint Language OCL. In Tok-Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, pages 449–464. Springer, Berlin, LNCS 1507, 1998. ISBN ISBN 3-540-65189-6. `http://www.db.informatik.uni-bremen.de/publications/Richters_1998_ER.ps.gz`.

[28] Mark Richters and Martin Gogolla. A Metamodel for OCL. In Robert France and Bernhard Rumpe, editors, *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, pages 156–171. Springer, Berlin, LNCS 1723, 1999. ISBN ISBN 3-540-66712-1. `http://www.db.informatik.uni-bremen.de/publications/Richters_1999_UML.ps.gz`.

[29] Mark Richters and Martin Gogolla. On the Need for a Precise OCL Semantics. In Robert France, Bernhard Rumpe, Brian Henderson-Sellers, Jean-Michel Bruel, and Ana Moreira, editors, *Proc. OOPSLA Workshop "Rigorous Modeling and Analysis with the UML: Challenges and Limitations"*. Colorado State University, Fort Collins, Colorado, 1999. `http://www.db.informatik.uni-bremen.de/publications/Richters_1999_OOPSLA-WS.ps.gz`.

[30] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, second edition, 1992. ISBN 013-978529-9. `http://spivey.oriel.ox.ac.uk/~mike/zrm/`.

[31] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science - Volume B: Formal Models and Semantics*. Elsevier, Amsterdam, first edition, 1990. ISBN 0-444-88075-5.

[32] Mandana Vaziri and Daniel 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.

[33] Jos Warmer and Anneke Kleppe. *The Object Contraint Language: Precise Modelling with UML*. Addison-Wesley Longman, Inc., Reading, MA, USA, 1999. ISBN 0-201-37940-6. `http://www.klasse.nl/ocl-boek/intro.htm`. This book covers only OCL 1.1.

[34] Jos Warmer, Anneke Kleppe, Tony Clark, Anders Ivner, Jonas Högström, Martin Gogolla, Mark Richters, Heinrich Hussmann, Steffen Zschaler, Simon Johnston, David S. Frankel, and Conrad Bock. Response to the UML 2.0 OCL RfP. Technical report, August 2001.

[35] Ralf Wiebicke. *Utility Support for Checking OCL Business Rules in Java Programs*. dilploma thesis, Technische Universität Dresden, December 2000. `http://rw7.de/ralf/diplom00/ocl-java.ps`.

[36] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993. ISBN 0-262-23169-7, 384 pages.

[37] Jim Woodock and Jim Davies. *Using Z*. Prentice Hall, 1996. ISBN 0-13-948472-8. `http://softeng.comlab.ox.ac.uk/usingz/`.