Electronic Communications of the EASST
Volume 5 (2006)

Proceedings of the Sixth OCL Workshop OCL for (Meta-)Models in Multiple Application Domains (OCLApps 2006)

Semantic Issues of OCL: Past, Present, and Future

Achim D. Brucker, Jürgen Doser, and Burkhart Wolff

20 pages

Guest Editors: Dan Chiorean, Birgit Demuth, Martin Gogolla, Jos Warmer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Semantic Issues of OCL: Past, Present, and Future

Achim D. Brucker, Jürgen Doser, and Burkhart Wolff

Information Security, ETH Zurich, 8092 Zurich, Switzerland
{brucker,doserj,bwolff}@inf.ethz.ch

Abstract: We report on the results of a long-term project to formalize the semantics of OCL 2.0 in Higher-order Logic (HOL). The ultimate goal of the project is to provide a formalized, machine-checked semantic basis for a theorem proving environment for OCL (as an example for an object-oriented specification formalism) which is as faithful as possible to the original informal semantics. We report on various (minor) inconsistencies of the OCL semantics, discuss the more recent attempt to align the OCL semantics with UML 2.0 and suggest several extensions which make, in our view, OCL semantics more fit for future extensions towards program verifications and specification refinement.

Keywords: HOL-OCL, UML/OCL, formal semantics

1 Introduction

In research communities, UML/OCL has attracted interest for various reasons:

1. it is a formalism with a "programming language face," which is perhaps easier to adopt by software developers notoriously hostile to mathematical notation,
2. it puts forward the idea of an object-oriented specification formalism, turning objects and inheritance into the center of the modeling technique, and
3. it provides in many respects a "core language" for object-oriented modeling which makes it a good target for research of object-oriented semantics.

Here, item 1 refers not only to syntax, but also to semantics: OCL semantics comprises the notion of undefinedness to model exceptional computations abstractly; this is deeply integrated into the logics and presents a particular challenge to deductive systems. Further, especially item 2 makes OCL rather different from logical languages such as first-order logics (FOL), higher-order logics (HOL), set theory and derived specification formalisms such as Z [WD96, BRW03] or VDM. Following a long platonic tradition in logics, these languages have a foundation in the notion of *values* and the definition of (hierarchies of) relations over them. This remarkably different perspective makes semantics for object-oriented specification difficult; numerous lukewarm attempts to integrate object-orientation into specification formalisms, such as VDM++ or Object-Z, report—among many useful things—on this particular difficulty. Comparing OCL with the two related approaches JML and Spec#, the main difference is that OCL attempts to abstract from concrete object-oriented programming languages, while JML and Spec# are designed as annotation-languages for them. This also holds for the UML Action package, which provides a deliberately abstract programming notation for "methods" associated to operation.

These three essentials motivated a long-term project to formalize the semantics of OCL 2.0 using HOL, leading to the proof environment HOL-OCL built on top of Isabelle/HOL [BW06a]

(http://www.brucker.ch/projects/hol-ocl/). The ultimate goal of the project is to provide calculi and automated proof support for reasoning over OCL formulae based on rules derived from this formalized semantics. This paves the way for proving the consistency of specifications, the proof-obligations resulting from specification refinements as well as the correctness of the transition to executable code. In this paper, we will present a by-product of this line of research: namely various formalization problems that we found or that we foresee when heading for an integrated verification method ranging from specifications to programming code. Extending earlier work [BW02], we report on a substantially larger range of problems and put it into perspective to recent developments of the OCL semantics.

2 Methodology: "Strong" Formal Semantics

In this section, we describe the foundations, the relevant techniques and the benefits of the methodology underlying HOL-OCL. This methodology boils down to provide a "strong," i. e., machine-checked and conservative, formalization of the "Semantics" chapter of the OCL standard [OMG03a, Appendix A]. The question may arise why the original formalization is not adequate for our goals. There are two reasons:

The fundamental reason results from the fact that [OMG03a, Appendix A] is based on naive set theory and an informal notion of "type" and "model." It assumes a universe for values and objects and algebras over it without any concern of existence and consistency. While a formalization in Zermelo-Fraenkel set theory and its implementation in Isabelle/ZF is probably feasible, it seems more natural to opt for a typed meta-language (like HOL comprising a typed set theory) since OCL is a typed language after all.

The technical reason is a consequence of our design choice to use the typed meta-language in a particular way, namely by mapping OCL types injectively to HOL types in our presentation. Consequently, only well-typed OCL formulae exist in HOL-OCL (others were rejected by Isabelle's type-checker), which makes well-formedness-related side-conditions in calculi unnecessary. Together with the fact that the Isabelle/HOL library can be re-used to a certain extent, this greatly improves the practicability of our approach.

Technically speaking, our representation is a so-called *shallow embedding* without an explicit datatype for syntax and an explicit semantic interpretation function I mapping syntactic terms to a semantic domain.

In the following, we present our meta-language HOL and the conservative methodology. We outline the shallow representation and show its equivalence to [OMG03a, Appendix A].

2.1 Higher-order Logic

Higher-order Logic (HOL) [Chu40, And86] is a classical logic with equality enriched by total parametrically polymorphic higher-order functions. It is more expressive than first-order logic, e. g., induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a typed functional programming language like Haskell extended by logical quantifiers.

HOL is based on the typed λ-calculus—i. e., the *terms* of HOL are λ-expressions. Types of terms may be built from *type variables* (like α, β, ..., optionally annotated by Haskell-like *type*

classes as in $\alpha :: order$ or $\alpha ::$ bot) or *type constructors* (like bool or nat). Type constructors may have arguments (as in α list or α set). The type constructor for the function space $\Rightarrow$ is written infix: $\alpha \Rightarrow \beta$; multiple applications like $\tau_1 \Rightarrow (\ldots \Rightarrow (\tau_n \Rightarrow \tau_{n+1})\ldots)$ have the alternative syntax $[\tau_1, \ldots, \tau_n] \Rightarrow \tau_{n+1}$. HOL is centered around the extensional logical equality $\_ = \_$ with type $[\alpha, \alpha] \Rightarrow$ bool, where bool is the fundamental logical type. We use infix notation: instead of $(\_ = \_)\ E_1\ E_2$ we write $E_1 = E_2$. The logical connectives $\_ \wedge \_$, $\_ \vee \_$, $\_ \rightarrow \_$ of HOL have type $[\text{bool}, \text{bool}] \Rightarrow$ bool, $\neg\_$ has type bool $\Rightarrow$ bool. The quantifiers $\forall \_.\_$ and $\exists \_.\_$ have type $[\alpha \Rightarrow \text{bool}] \Rightarrow$ bool. The quantifiers may range over types of higher order, i. e., functions or sets.

The type discipline rules out paradoxes such as Russel's paradox in untyped set theory. Sets of type α set can be defined isomorphic to functions of type $\alpha \Rightarrow$ bool; the definition of the elementhood $\_ \in \_$, the set comprehension $\{\_.\_\}$, $\_ \cup \_$ and $\_ \cap \_$ is then standard.

The *modules* of larger logical systems built on top of HOL are Isabelle *theories*. Among many other constructs, they contain type and constant declarations as well as axioms. Since stating arbitrary axioms in a theory is extremely error-prone and should be avoided, only very limited forms of axioms should be admitted and the side-conditions (both syntactical and semantical) checked by machine. These fixed blocks of declarations and axioms described by a syntactic scheme are called *conservative theory extensions* since any extended theory is consistent ("has models") provided the original theory was. Most prominent in the literature are *constant definition* and *type definition*. For example, a constant definition consists of a declaration declaring constant c of type τ and the (well-typed) axiom of the form: $c = E$ with the side-condition that c has not been previously declared, E does neither contain free variables nor c (no recursion). A further side-condition forbids type variables in the types of constants in E that do not occur in the type τ. As a whole, a constant definition can be seen as an "abbreviation," which makes the conservativity of the construction plausible (see [GM93] for details). The idea of an "abbreviation" is also applied to the conservative *type definition* of a type $(\alpha_1, \ldots, \alpha_n)T$ from a set $\{x \mid P(x)\}$.

The entire Isabelle/HOL library, including typed set theory, well-founded recursion theory, number theory and theories for data-structures like pairs, type sums and lists is built on top of the HOL core-language by conservative definitions and derived rules. This methodology is also applied to HOL-OCL.

2.2 Formal Semantics Preliminaries in HOL

In OCL, the notion of explicit undefinedness plays a fundamental role, both for the logical and non-logical expressions:

> Some expressions will, when evaluated, have an undefined value. For instance, typecasting with `oclAsType()` to a type that the object does not support or getting the `->first()` element of an empty collection will result in undefined.
>
> *(OCL Specification [OMG03a], page 15)*

Thus, concepts like *definedness* and *strictness* play a major role in the OCL. We use a *type class* bot to specify the class of all types that contain the undefinedness element $\bot$. For all types in

this class, we define a combinator strictify by:

$$\text{strictify}\, f\, x \equiv \text{if}(x = \bot)\, \text{then}\, \bot\, \text{else}\, f\, x$$

with type $(\alpha :: \text{bot} \Rightarrow \beta :: \text{bot}) \Rightarrow \alpha \Rightarrow \beta$. The operator strictify yields a strict version of an arbitrary function f.

Further, we use the type constructor $\tau_\bot$ that assigns to each type τ a type *lifted* by $\bot$. Per construction, each type $\tau_\bot$ is in fact in the type class bot. The function $\lfloor\_\rfloor : \alpha \to \alpha_\bot$ denotes the injection, the function $\lceil\_\rceil : \alpha_\bot \to \alpha$ its inverse for defined values.

On the expression level, lifting combinators defining the distribution of *contexts* or *environments* (see below) are defined as follows:

$$\begin{aligned}
&\text{lift}_0\, f \equiv \lambda\, \tau.\, f && \text{of type } \alpha \Rightarrow V_\tau(\alpha)\,,\\
&\text{lift}_1\, f \equiv \lambda\, X\, \tau.\, f(X\, \tau) && \text{of type } (\alpha \Rightarrow \beta) \Rightarrow V_\tau(\alpha) \Rightarrow V_\tau(\beta)\,, \text{and}\\
&\text{lift}_2\, f \equiv \lambda\, X\, Y\, \tau.\, f(X\, \tau)(Y\, \tau) && \text{of type } ([\alpha, \beta] \Rightarrow \gamma) \Rightarrow [V_\tau(\alpha), V_\tau(\beta)] \Rightarrow V_\tau(\gamma)\,.
\end{aligned}$$

where $V_\tau(\alpha)$ is a synonym for $\tau \Rightarrow \alpha$. The types of these combinators reflect their purpose: they "lift" operations from HOL to semantic functions that are operations on contexts.

2.3 Textbook vs. Combinator Style Semantics of Operations

In HOL-OCL, we use a combinator-style presentation of the semantic functions rather than a textbook-style presentation as used in the OCL standard, both for reasons of conciseness as well as accessibility to advanced techniques of automatic generation of library theorems [BW03]. In combinator style as used in the HOL-OCL libraries, for example, the constant 1, the unary operation `not _`, and the binary operation $\_ + \_$ are represented by the following constant definitions:

$$\begin{aligned}
1 &\equiv \text{lift}_0(\lfloor 1 \rfloor)\\
\texttt{not}\ _ &\equiv \text{lift}_1(\text{strictify}(\lfloor_\rfloor \circ (\neg_) \circ \lceil_\rceil)\\
_ + _ &\equiv \text{lift}_2(\text{strictify}(\lambda\, x.\ \text{strictify}(\lambda\, y.\ \lfloor \lceil x \rceil + \lceil y \rceil \rfloor)))
\end{aligned}$$

where $\_ \circ \_$ denotes function composition. We use overloading here: the $\_ + \_$ on the left-hand side of the last definition has type $[V_\tau(\text{int}_\bot), V_\tau(\text{int}_\bot)] \Rightarrow V_\tau(\text{int}_\bot)$ (where $V_\tau(\text{int}_\bot)$ is the HOL equivalent to the OCL type `Integer`), while the $\_ + \_$ on the right-hand-side has type $[\text{int}, \text{int}] \Rightarrow \text{int}$. This definition directly translates the idea that $\_ + \_$ in HOL-OCL is the strictified version of the "mathematical" $\_ + \_$ lifted over contexts.

The question arises why this definition is equivalent to the formalized version of the semantics given in the standard. The OCL 2.0 standard presents a definition scheme for all *strict basic operations* by just one example. For the $+$-operator on integers, [OMG03a, page A-11] presents this definition as:

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

This semantic function for basic operations is integrated in the more general semantic interpretation function for OCL expressions like

> 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.
> iv. $I[\![w(e_1, \ldots e_n)]\!]\tau = I(w)(\tau)\big(I[\![e_1]\!](\tau), \ldots, I[\![e_n]\!](\tau)\big)$
>
> *(OCL Specification [OMG03a], page A-26, definition A.30)*

Here, τ refers to the environment (in the sense of the standard), i. e., a pair consisting of a map β assigning variable symbols to values and a pair σ of system states.

There are two more semantic interpretation functions; one concerned with path expressions (i. e., *attribute and navigation expressions* [OMG03a, Definitions A.21], and one concerning the interpretation of pre and postconditions $\tau \vDash P$ which is used in two different variants.

To show the equivalence of the two formalization styles, we introduce a "explicit semantic function" $I[\![E]\!]\tau$ into our shallow embedding as a syntactic marker, i. e., by stating the identity:

$$I[\![x]\!] \equiv x \qquad \text{with type } \alpha \Rightarrow \alpha\,.$$

For the addition over `Integer`, we prove the following theorem that explicitly states that our defined operator is an instance of the informal definition scheme in the standard:

$$I[\![X + Y]\!]\tau = \begin{cases} \lfloor \ulcorner I[\![X]\!]\tau \urcorner + \ulcorner I[\![Y]\!]\tau \urcorner \rfloor & \text{if } I[\![X]\!]\tau \neq \bot \text{ and } I[\![Y]\!]\tau \neq \bot, \\ \bot & \text{otherwise}\,. \end{cases}$$

The proof in HOL-OCL is simple and canonical: it consists of the unfolding of all combinator definitions and the syntactic marker I. The combinators are just abbreviations of re-occurring patterns in the textbook style definitions.

In the following, we summarize the differences between the OCL standards textbook definitions and our combinator-style approach:

1. The standard [OMG03a, chapter A] assumes an *untyped set of values and objects* as semantic universe of discourse. Since we reuse the types from the HOL-library to give Booleans, Integers and Reals a semantics, meta-expressions like $\{\mathrm{true}, \mathrm{false}\} \cup \{\bot\}$ used in the standard are simply illegal in our interpretation. This makes the injections $\lfloor\_\rfloor$ and projections $\ulcorner\_\urcorner$ necessary.
2. The semantic functions in the standard are split into $I(x)$, $I[\![e]\!]\tau$, $I_{\mathrm{ATT}}[\![e]\!]\tau$ and $\tau \vDash P$. Since we aim at a shallow embedding (which ultimately suppresses the semantic interpretation function), we prefer to fuse all these semantic functions into one.
3. The *environment* τ in the sense of the standard is a pair of a variable map and a pair of pre and post state. The variable map is superfluous in a shallow embedding (binding is treated by using higher-order abstract syntax), our contexts τ just consist of the state pair.

Of course, this presentation here covers only one aspect of the compliance of the HOL-OCL semantics to the standard for a tiny portion of the language; for an in-depth discussion for the complete language, the reader is referred to [BW06a].

2.4 The Benefits of a "Strong" Formal Semantics

Our strong formalization of [OMG03a, Appendix A] has the following benefits:

A Consistency Guarantee. Since all definitions in our formal semantics are conservative and all rules are derived, the consistency of HOL-OCL is reduced to the consistency of HOL for the *entire language*.

A Technical Basis for a Proof-Environment. Based on the derived rules, control programs (i. e., *tactics*) implement automated reasoning over OCL formulae; together with a compiler for class diagrams, this results in a general proof environment called HOL-OCL. Its correctness is reduced to the correctness of a (well-known) HOL theorem proving system.

Proofs for Requirement Compliance. The OCL standard contains a collection of formal requirements in its mandatory part with no established link to the informative part [OMG03a, Appendix A]. We provide formal proofs for the compliance of our OCL semantics with these requirements (see [BW06a] for details).

Formalization Experience. Since our semantics is machine-checked, we can easily change definitions and check properties of them allowing for increased knowledge of the language as a whole.

3 The Past

OCL standards are developed in an open process, led by the OMG (Object Management Group). This process leads to a variety of (intermediate) "standardization" documents. This is especially true for UML and OCL, which have a long history. OCL was introduced as an OMG specification language as an additional document [OMG97] completing the UML 1.1 standard [OMG99]. In later releases of the UML standards of the version 1.x series the OCL standard was a chapter of the UML specification, e. g., [OMG03b, Chapt. 6].

All the different versions of OCL 1.*x* are very close to each other, containing mainly an informal motivation of the indented use and semantics[1] of OCL together with a formal grammar of its concrete syntax. Understandably, these past version of the standard lacked many desirable features, e. g., the use of OCL was mainly limited to annotate class diagrams, no abstract syntax was included. Moreover, reading the OCL 1.*x* standards leaves more questions open than it answers. These shortcomings and open questions, like the handling of undefinedness, or recursion, were discussed [VJ00, MC99, HCH+98, CKM+02] in academia and this discussions clearly fertilized the development towards OCL 2.0. Especially the work of Richters [Ric02] in developing a formal semantics served as formal underpinning of the OCL 2.0 development. It was a major break-through in the process of defining a formal semantics for OCL. Many problems, like the handling of undefinedness, were clarified during the OCL 2.0 standardization process, some questions however, like the handling of recursion, are still unsolved.

4 The Present

4.1 The OCL 2.0 Standard

In this section, we give a brief overview of the chapters of the standard that are related with the semantics of OCL 2.0: first, the OCL standard is divided into *normative* parts and *informa-*

[1] A good overview of the different usages of the word "semantics" is given in [HR04].

tive, i. e., not normative, parts. The semantics of the standard appears in the following chapters of [OMG03a]:

Chapter 7 "OCL Language Description": This *informative* chapter motivates the use of OCL and introduces it informally, mostly by examples.

Chapter 10 "Semantics Described using UML": This *normative* chapter describes the "semantics" of OCL using the UML itself. Merely an underspecified "evaluation" environment is presented.

Chapter 11 "The OCL Standard Library": This *normative* chapter is, in our opinion, the best source of the normative part of the standard describing the intended semantics of OCL. It describes the semantics of the OCL expressions as requirements they must fulfill.

Appendix A "Semantics": This *informative* appendix, based on [Ric02] , defines the syntax and semantics of OCL formally in a textbook style paper-and-pencil notion.

We see the semantic foundations of the standard critical for several reasons:

1. The normative part of the standard does not contain a formal semantics of the language.
2. The consistency and completeness of the formal semantics given in "Appendix A" is not checked formally.
3. There is no proof, neither formal nor informal, that the formal semantics given in the informative "Appendix A" satisfies the requirements given in the normative chapter 10.

Nevertheless, we think the OCL standard [OMG03a] ("`ptc/03-10-14`") is mature enough to serve as a basis for a machine-checked semantics and formal tools support. More recent versions, especially [OMG06] ("`formal/06-05-01`"), are an ad-hoc attempt to align the UML 2.0 with the OCL 2.0. Among many other annoyances, new datatypes are introduced without giving them a consistent semantics. For example, besides `OclUndefined` (called `invalid`), [OMG06] introduces a second exception element called `null`. On the one hand, the intention of the authors of the standard is to give `OclInvalid` a strict and `null` a non-strict semantics with respect to collection type constructors: "Note that in contrast with `OclInvalid` `null` is a valid value and as such can be owned by collections." [pp. 36][OMG06]. Nevertheless, `null` is still strict with respect to other operations: "Any property call applied on null results in OclInvalid, except for the operation oclIsUndefined()" [pp. 138][OMG06]. On the other hand, both `invalid` and `null` conform to all classifiers, in particular `null` conforms to `invalid` and vice versa. Moreover, the conforms relationship is antisymmetric and therefore `invalid` and `null` are actually indistinguishable. Considering that many of these changes were made without giving much thought to their impact on the existing specification, [OMG06] represents a considerable step back with respect to consistency and potential for formal semantics. In particular, all issues addressed in this paper are also valid for [OMG06].

In the remainder of this section, we will explain some selected problems grouped into the three categories 1. data and object structures, 2. built-in operations, and 3. user-defined operations. Our choice of "issues" focuses on semantical problems which are caused either by conceptual ambiguities, by plain inconsistencies between the various parts of the standard document or by missing concepts. Missing concepts not considered vital are labeled as extensions and discussed in the next section.

4.2 Data- and Object Structures

Our most foundational criticism of the semantics chapter [OMG03a, Appendix A] is its use of naive set theory as basis for the notions "type," "state" and "model". For example, types were explained by some type interpretation function $I(t)$ [OMG03a, introduced in Definition A.14] mapping to a (never described) universe of values and objects. The expression interpretation function $I[\![E]\!]$ assumes that variables and key operator symbols have been annotated with type expressions like in $\_ =_t \_$. Therefore typing is a prerequisite of the semantic construction of OCL. $I[\![E]\!]$ uses these type annotations to project and inject into subsets of the universe described by $I(t)$ without proof or argument that these definitions actually respect the typing. Also, the standard does not specify what correct typings are (the semantic function is defined for arbitrary typings), whether they are unique, and how to derive them. Using a typed semantic domain resolves this problems automatically, as we will see in Section 4.2.2.

In the following discussion, we use the notation of the standard $(\sigma_{\text{pre}}, \sigma_{\text{post}}) \vDash E$, meaning $I[\![E]\!]((\sigma_{\text{pre}}, \beta), (\sigma_{\text{post}}, \beta)) = \text{true}$ for all variable assignments β. We also abbreviate the state-pair $(\sigma_{\text{pre}}, \sigma_{\text{post}})$ with the "context" τ.

4.2.1 Referential and Non-Referential Object Universes

The following principles on the object universe were stated informally in

> Each object is uniquely determined by its identifier and vice versa.
>
> *(OCL Specification [OMG03a], page pp. A-7, A.1.2.1)*

From the perspective of a *state* of a system, i. e., a partial map from object-id's (*oid*'s) to the object universe, there are essentially two ways to achieve this: Either we assume *global memoization* semantics, i. e., a constructor looks up the complete state when creating a new object and yields, if existent, the oid to the formerly stored object, or we assume that the oid is actually stored inside the object and "fresh" at object creation. In more detail, objects with attributes $a_1, \ldots a_n$ of OCL types $T_1, \ldots, T_n$ are represented semantically either as tuples of type $\tau_1 \times \ldots \times \tau_n$ where τ_i is the corresponding HOL type of T_i or by tuples of type $\text{oid} \times \tau_1 \times \ldots \times \tau_n$. We call the latter alternative a "referential object universe" and the former, consequently, a "non-referential object universe".

Since global memoization semantics reveals unusual behavior, for example with respect to the `oclIsNew()`-operation, we suggest to assume the referential universe; the one-to-one correspondence between objects and references allows then to represent a class type by the set of object representations and not by a set of `oid`'s, i. e., $I(C) = \{x :: \text{oid} \times \tau_1 \times \ldots \times \tau_n \times \alpha.\ \text{true}\}$ for a class type C. With the type variable α, we can summarize all subtypes, i. e., future extensions of this object representation.

Thus, the typing of objects can be handled at the meta-level instead of some predicative constraints in ZF. Subtyping in OCL can be expressed by type variable instantiation in HOL (note, however, that the presentation here is slightly simplified; a full-blown discussion of our construction can be found in in [BW06a]). A useful by-product of this object presentation is a universal `_@pre`-operator that takes the `oid` of an object representation and dereferences it in

the pre-state; this substantially simplifies the path expression semantics and makes statements such as `self = self@pre` (crucial for frame properties, see Section 5.1.5) meaningful.

4.2.2 Types

The OCL language is typed; the type annotation syntax `E:T` for all expression `E` and all type expressions `T` is the visible display of this fact. The standard semantics definition uses this annotations inherently for variable and operator symbol annotations; and as we said, we find their role not adequately described. When using a typed meta-logic, this means that we can "implement" `E:T` by $E :: T$, i. e., the built-in type annotation of HOL (where T denotes the HOL type corresponding to `T`). As a consequence, the construction of type-annotation is formally defined (by the HOL type-system), implemented by well-understood and fast type-inference algorithms (the Hindley-Milner algorithm), and, last but not least, solves some missing foundational links: Whenever given a semantic definition of an operation in a typed setting, its typecheck will assure the semantically consistent treatment of typing. Furthermore, whenever we derive a rule over typed presentations, we also prove the subject reduction property for this rule implicitly.

For example, if we define the default equality $\_ \doteq \_$ by:

$$\_ \doteq \_ \equiv \mathrm{lift}_2(\mathrm{strictify}(\lambda\, x.\ \mathrm{strictify}(\lambda\, y.\ \lfloor \ulcorner x \urcorner = \ulcorner y \urcorner \rfloor)))$$

i. e., by using the HOL equality $\_ = \_ :: [\alpha, \alpha] \Rightarrow \mathit{bool}$, the typecheck of this definition already assures the type-consistent use of semantic values throughout this semantic definition.

The astute reader will notice that type-construction via Hindley-Milner can not make *all* implicit type-coercions of OCL explicit; consider $(a : A) \doteq (b : B)$ where B is subtype of A. Type inference will not succeed for this form, but it will for $a \doteq b$ `.oclAsType(`A`)`, or, of course, for a `.oclAsType(OclAny)`$\doteq b$ `.oclAsType(OclAny)`. We argue that this part of making coercion explicit can be treated as *simple* syntactic pre-processing and that it is conceptually independent from the core of the semantics definition.

Type annotations carry semantic information, which is also important in deduction. For example, consider that we would like to state the commutativity of addition:

$$\tau \vDash x + y \triangleq y + (x :: \mathit{Real})\,,$$

which is a theorem in HOL-OCL. In an un-typed universe, this statement cannot be interpreted: we have to annotate the types on the object-level first, e. g., for each variable and the $\_ + \_$-operator. Note that the HOL-typing influences the validity of OCL formulas and has therefore also consequences in a logical derivation:

$$\tau \vDash (x :: \mathit{Real})\ \texttt{.IsTypeOf(Real)}$$

is a general theorem in HOL-OCL. Thus, explicit type annotations can in many cases be eliminated, which simplifies proofs.

4.2.3 Types and Class Invariants

The standard makes actually a very strong requirement for the connection between types (resulting from a class) and class invariant:

> When the invariant is associated with a Classifier, the latter is referred to as a "type" in this chapter. An OCL expression is an invariant of the type and must be true for all instances of that type at any time.
>
> *(OCL Specification [OMG03a], page Chapter 7.3.3,pp 8)*

A later statement in chapter 12.6 essentially rephrases this requirement.

For defined objects, it is therefore hard to escape that a semantics for UML should enjoy the following relations between syntactic type, semantic type and the invariant inv_A of a class A:

$$\tau \vDash E\,\texttt{.oclIsKindOf}(A)\ \texttt{implies}\ inv_A(E)$$
$$\tau \vDash A\texttt{::allInstances()->includes}(E)\ \texttt{implies}\ E\,\texttt{.oclIsKindOf}(A)$$
$$\tau \vDash E.a\,\texttt{.oclIsUndefined()}\ \texttt{implies}\ E.a\,\texttt{.oclIsKindOf}(A)$$

Unfortunately, [OMG03a, Appendix A] only partially achieves this goal; not all expressions of a class type occuring in specifications denote instances in the sense above. In order to understand this, we reproduce the standards definition of an accessor function (note that we glue several scattered definitions together in order to simplify the presentation):

$$I[\![E.a^{(1)}]\!](\sigma_{\text{pre}},\sigma_{\text{post}}) = \text{if } I[\![E]\!](\sigma_{\text{pre}},\sigma_{\text{post}}) \in \text{dom}(\sigma_{\text{post}}) \text{ then } (\sigma_{\text{post}})_{\text{ATT}}(a) \text{ else } \bot$$

We refer to the underlying semantics of this accessor definition, i. e., $E.a^{(1)}$, as *structural*. Then, when introducing invariants, the concept of a *valid state* σ is defined via satisfaction of the invariants:

$$\begin{aligned}\sigma \vDash\ & C_1\texttt{::allInstances()->forall}(x_1 : C_1 |\\ & \cdots\\ & \quad C_n\texttt{::allInstances()->forall}(x_n : C_n |\\ & \qquad inv_{C_1}\ x_1\ \texttt{and}\ \ldots\ \texttt{and}\ inv_{C_n}\ x_n)\cdots)\end{aligned}$$

Thus, the third from the postulates above does not hold in general, but only for valid system states.

We call an OCL invariant a *structural invariant* if all accessors inside have structural semantics. An alternative to this, namely *semantic invariants*, consists in:

1. We define co-inductively the set of all "valid" objects of type A respectively for kind A,
2. we interpret *self* `.IsTypeOf`(A) respectively *self* `.oclIsKindOf`(A) in these sets, and
3. we use *semantic* accessors in invariants, preconditions and postconditions.

As a *semantic accessor*, i. e., $E.a^{(2)}$, we define a structural accessor, that additionally checks if the resulting structural object actually satisfies the invariant. If not, it returns $\bot$:

$$I[\![E.a^{(2)}]\!]\ \tau = \text{if } inv_A(I[\![E.a^{(1)}]\!]\ \tau \text{ then } I[\![E.a^{(1)}]\!]\ \tau \text{ else } \bot$$

With semantic accessors, we also achieve the third from the three postulates above. The resulting notion of invariants, preconditions and postconditions is stronger than the original one. Under certain minor side-conditions the semantic invariant is equivalent to the structural one for a valid system state σ; thus, the original formulation can, for example, still be used for runtime-testing.

In our experience, proofs over cyclic object structures, like linked lists, become significantly easier with semantic invariants, since rules like

$$\frac{\tau \vDash \texttt{not } self \texttt{ .oclIsUndefined()}}{\tau \vDash inv_{\text{Node}}(self) \texttt{ implies } inv_{\text{Node}}(self.\text{next}^{(2)})}$$

can be derived. Moreover, strong invariants open up new ways for proof automation based on co-induction.

4.2.4 Smashed Datatypes

The OCL standard defines all operations as strict, i. e., the evaluation of an operation is undefined if one of its argument is undefined. In particular, constructors of collection types such as sets or of tuple types are strict. However, in the informative part [OMG03a, Appendix A], semantic values are included in the semantic universe that can actually not be denoted in the language, like $\{\bot, 1\}$. In other words, the semantics is not fully abstract.

We suggest to identify these structures that *contain* $\bot$ with $\bot$ itself; In the literature, such sets (or generic data-structures) with this quotient construction are called *smashed*. The quotient introduces congruences like $(\bot, X) = \bot$ or $\{a, \bot, b\} = \bot$. This semantics is characterized by the derived rule for all collection types:

$$\frac{\tau \vDash self\texttt{->includes}(e)}{\tau \vDash \texttt{not } e \texttt{ .oclIsUndefined()}}$$

We strongly opt for a smashed collection semantics, mainly for three reasons:

1. OCL tends to define its constructs towards executability and proximity to object-oriented programming languages such as Java, where constructors are strict. Introducing lazy constructors is therefore no option,
2. smashed semantics reflects strict constructors, and
3. OCL with non-smashed collection semantics leads to very un-natural calculi.

With respect to the last item, consider for example the vital `forAll`-introduction rule

$$\frac{\tau \vDash \texttt{not } self \texttt{ .oclIsUndefined()} \qquad \begin{array}{c}[\tau \vDash self\texttt{->includes}(x)] \\ \vdots \\ \tau \vDash P(x)\end{array}}{\tau \vDash self\texttt{->forall}(x|P(x))}\ .$$

This rule only holds for a smashed collection semantics. Moreover, even statements on sets that look obvious on the first sight, and are also requirements of the standard, like

$$\tau \vDash \texttt{not } self \texttt{ .oclIsUndefined()} \wedge \texttt{not } x \texttt{ .oclIsUndefined()} \\ \texttt{implies } self\texttt{->including}(x)\texttt{->includes}(x)$$

and

$$\tau \vDash \texttt{not}\ \textit{self}\ \texttt{.oclIsUndefined()} \wedge \texttt{not}\ x\ \texttt{.oclIsUndefined()} \\ \texttt{implies}\ \textit{self}\texttt{->count}(x) <= 1$$

do also only hold for a smashed collection semantics.

4.3 Built-in Operations

4.3.1 Implies

Recall that the OCL logic is based on a strong *Kleene Logic*. Consequently, most operators of the logical type like `_ and_` are explicitly stated exceptions from the "operations are strict"-principle. In this section, we will discuss the `implies` operation in more detail. Its semantic is defined in the standard as follows:

1. [OMG03a, Chapter 11] requires the following property of implies:

```
context Boolean::implies(b:Boolean):Boolean
  post: (not self) or (self and b)
```

2. [OMG03a, Appendix A] defines the `b1 implies b2` by a truth table, resulting in the "classical definition" of implication:

```
context Boolean::implies(b:Boolean):Boolean
  post: (not self) or b
```

These definitions contradict each other in the case where `self` is $\bot$ and `b` is `true`. Whereas different variants for implications for three-valued logics are considered in the literature [Häh91, HHB02], an analysis of the consequences for proof calculi reveals some bad surprises. For example, consider the usual assumption rearrangement rules valid in the "classical definition":

$$((X\ \texttt{or}\ \ Y)\ \texttt{implies}\ Z) = (X\ \texttt{implies}\ Z)\ \texttt{and}(Y\ \texttt{implies}\ Z)$$
$$((X\ \texttt{and}Y)\ \texttt{implies}\ Z) = (X\ \texttt{implies}\ (Y\ \texttt{implies}\ Z))$$
$$X\ \texttt{implies}\ (Y\ \texttt{implies}\ Z) = Y\ \texttt{implies}\ (X\ \texttt{implies}\ Z)$$

which *do not hold* for the standard's definition of the implication. Although the choice made in the normative part of the standards is feasible, it invalidates many crucial deduction techniques. In the light of these dramatic algebraic deficiencies, we qualify it as glitch from the deduction point of view and suggest to apply the definition used in the appendix.

4.3.2 Equalities

We have already seen that the standard semantics defines (informally) the equality as the strict extension of the equality of the underlying set theory [OMG03a, Appendix A]. For values, this implies *value equality*, for class objects which were represented by references *referential equality* (see [KC86] for an overview over the variety of different "equalities" object-oriented systems can be equipped with).

We suggest to add besides the default equality a strong equality defined by:

$$\_ \mathrel{\hat{=}} \_ \equiv \mathrm{lift}_2((\lambda\, xy.\, \lceil x \rceil = \lceil y \rceil))$$

Strong equality can actually be defined in OCL in terms of strict equality and `oclIsUnDefined()`.

The difference between strict and strong equality is that $o \doteq \bot$ evaluates to $\bot$ while, e. g., $\bot \mathrel{\hat{=}} \bot$ evaluates to *true*. Therefore, we can state rules like:

$$(o \doteq \bot) \mathrel{\hat{=}} \mathtt{true}$$

Moreover, strong equality enjoys the congruence properties reflexivity, symmetry, transitivity and substitutivity; these properties are a pre-requisite for term-rewriting and therefore of outstanding importance for automated reasoning. Further, users may *want* to use strong equality explicitly. For example, consider the following operation specification:

```
context C::m(a:Integer):Integer
post: result = 5 div a
```

What is the semantics of this operation given that the precondition does not rule out `a=0`? If the standard strict equality is used this results in an inconsistent specification. If the strong equality is used this operation simple returns undefined when called with an argument of `0`. Depending on the circumstances, both may be reasonable.

4.4 User-defined Operations

4.4.1 Overloading and Late Binding

The concept of method-overloading is not yet fully supported by OCL. We believe, this is more or less due to some accidental circumstances:

1. The UML standard [OMG03b, chapter 4.4.1] requires that operation names are unique within the same namespace. In particular, subclasses may not overwrite inherited operations. Albeit, the UML standard allows one to (explicitly) overwrite *methods*, i. e., *implementations* of operations.
2. The OCL standard [OMG03a, chapter 7.3.41] restricts the use of the precondition and postcondition declarations to operations or other behavioral features. Sadly, all OCL tools we know of do not support the specification of preconditions and postconditions for methods.
3. While the OCL standard speaks in several places of operation calls, it does not give an hints how operation overloading should be resolved, neither does it explain in detail concepts like operation (method) calls or operation (method) invocations.

Bringing these items together, one has to conclude that operation overloading, and thus late-binding, is underspecified, or even not supported in OCL. Nevertheless, we think that overwriting inherited operations or methods is a very important feature of object-orientation and should be supported by the OCL: since operation calls can occur in OCL constraints, their meaning depends on the semantics of operation invocation. Thus we provide the theoretical foundations for supporting late-binding (and thus overloading of operations) within HOL-OCL [BW06a], nevertheless a concrete syntax for specifying this has to be worked out. As simple workarounds, one can ignore the well-formedness constraint of UML for operations that requires operation names to be unique within one namespace. This is what most case-tools do.

Since the UML definition expresses in several places a clear preference for overloading operations, we suggest to extend the current OCL standard by a late-binding semantics of method invocation. We are aware that checks for conservativity will impose restrictions on invocations here to be discussed in Section 4.4.2.

4.4.2 Recursion

The OCL standard vaguely requires that recursions should always be terminating to rule out problems with divergent operation invocations:

> The right-hand-side of this definition may refer to operations being defined (i. e., the definition may be recursive) as long as the recursion is not infinite.
> *(OCL Specification [OMG03a], page paragraph 7.5.2, pp.16)*

and also:

> 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 [...]. However, allowing recursive operation calls considerably adds to the expressiveness of OCL. We therefore allow recursive invocations as long as the recursion is finite. Unfortunately, this property is generally undecidable. *(OCL Specification [OMG03a], page A-31)*

Unfortunately, in a proof-environment we have to be substantially more specific than this. Furthermore, HOL-OCL is designed to live with the open-world assumption, i. e., with the potential extensibility of object universes, as a default; further restrictions such as finalizations of class diagrams or a limitation to Liskov's Principle [LW94] may be added on top, but the system in itself does not require them. This has the consequence that even in the following example:

```
context C::m(a1:T1,...,an:Tn):Integer
  post: result = if a1.p()
                 then 1 + self.m(a1.q(),...,an)
                 else 0 endif
```

the termination for the invocation `self.m(a1.q(),...,an)` is fundamentally unknown (even if `p` and `q` are known and terminating): a potential overriding may destroy the termination of this recursive scheme.

In form of a pre-translation process, operation specifications with a limited form of recursive invocations can be defined by a finite family of constant definitions. These limited forms can be listed as follows:

- calls to superclass operations, i. e., `(self.asType(A)).m(x1,...,xn)`, or
- direct recursive well-founded invocations, i. e., `(self.asType(C)).m(x1,...,xn)` where the user specifies a "measure" or a well-founded ordering which the system checks to be respected in all calls.

The first can be statically resolved, the latter is based on the theory of well-founded orders and the well-founded recursor "wfrec" in Isabelle/HOL, and so to speak an application of the standard HOL methodology to OCL.

Alternatively, in case of a finalized class, i. e., a class that cannot be further extended by inheritance, late-binding can be can by replaced by a case-switch. For example, consider the case of three classes A, B, and C, related by inheritance:

$$\begin{array}{l}\text{if } self\texttt{.IsTypeOf}(A) \text{ then } S\\ \text{else if } self\texttt{.IsTypeOf}(B) \text{ then } S'\\ \text{else if } self\texttt{.IsTypeOf}(C) \text{ then } S''\\ \text{else } \bot\end{array}$$

where S, S', and S'' are the corresponding operation specifications. Summing up, conservativity implies that only limited forms of recursive invocations are admissible. In an open world (no class finalization so far), only operation invocations on objects can be treated, whose type has been fixed, in a closed world (the class hierarchy has been finalized), an invocation can be expanded to a case-switch considering the dynamic type of *self* over calls.

5 The Future

Future extensions will aim for allowing smooth transitions from specification to code ("methods" implementing "operations" in the terminology of the standard). There are various aspects of this global challenge, which are in parts already discussed in research communities as well as in the standardization process.

5.1 Library Extensions

5.1.1 Strict logical operators

The standard explicitly requires the logical operators as non-strict; in particular, they obey the laws of a strong Kleene-logic. However, due to the interplay between strict collection operators such as union and this non-strict logical operators, it is perhaps advisable to have strict logical operators too. For example:

```
A->union(B)->includes(a) = A->includes(a) sor B->includes(a)
A->intersect(B)->includes(a) = A->includes(a) sand B->includes(a)
```

The strict versions of the logical connectives have also a number of pragmatical advantages: they can be mapped to more efficient code, and sometimes a designer may *want* to express a certain property explicitly by them.

5.1.2 Machine Numbers

We suggest to introduce *bounded* versions of Integers with a concrete machine arithmetic based on a two's complement model (e. g., as described in [GJSB00]). This would allow a transition

to implementation languages, e. g., Java. These machine arithmetics are somewhat more distant to mathematics (MaxInt $+ 1 =$ MinInt, all commutative ring properties hold except $a - a = 0$, $0 \leq \text{abs}(a)$ does not hold for all defined a, etc.). But this behavior is a widely used implicit standard in microprocessor technology. Verification of such transitions from mathematical integers to machine integers can be a real concern in safety-critical applications, therefore it would be beneficial to have machine integers in the language. For a concrete proposal of a "strong" formalization of the Java arithmetics, see [RW03].

5.1.3 Infinite Collection Types

It is conceivable to drop the standards limitation to *finite* collection types. Infinite sets are clearly a very powerful and useful specification means which would allow to explicitly use the infinite sets occurring implicitly in OCL (such as the set of Integers), e. g., by quantifying over them. But there are also useful applications for infinite sequences and bags: They pave the way for new forms of recursion. A recursive method over an object-structure collecting the sequence of values of an attribute does not necessarily have to terminate: semantics could be defined via co-recursion. For code-generators, this means that lazy evaluation techniques known from functional programming must be applied. In HOL-OCL, sets can be infinite by default; as a consequence, for each type, there can be a constant that denotes the set of all elements of this type. We suggest to extend OCL with an operation `typeSetOf()` for accessing these type sets. For example, then one can state properties like `Integer::typeSetOf()->forAll(x|x+y=y+x)`. Note that a query for being finite can be expressed by asking if the cardinality of this set is defined, i. e., we can easily introduce a new operation `isFinite()`:

```
context isFinite(S:Set(T)):Boolean
  post: result = S->size().oclIsDefined()
```

5.1.4 References and Referential Types

In a referential universe, expressing the one-to-one "correspondance" between object representations and `oid` directly, it is also possible to define a referential type $Ref(A)$ of `oid`'s that map to objects of class A. Technically, the type is defined over the typed set:

$$\text{Ref}(A) = \text{oidOf}\,`\,A$$

where oidOf is just the projection of an object to its *oid*, $_\,`\,\_$ the pointwise map and $A$ the set of object-representations of OCL type `A`.

This extension also has advantages for programming languages used for implementing methods (see IMP++ for example in [BW06b], which also contains a verification technique). Since the relation between object representations and `oid` is a "one-to-one correspondance", it is straight forward to define a reference operator `&x` (like e. g., in C++). Further, the assignment operator assigning a reference to an attribute of type oid can then also be realized easily.

5.1.5 Frame-properties

The OCL does not guarantee that an operation only modifies the path-expressions mentioned in the postcondition, i. e., it allows arbitrary relations from pre-states to post-states. For most applications this is too general: there must be a way to express that parts of the state *do not change* during a system transition, i. e., to specify the frame properties of system transition. Thus we suggest to extend OCL to specify the frame properties explicitly:

```
(S:Set(OclAny))->modifiedOnly():Boolean
```

where `S` is a set of objects (i. e., a set of `OclAny` objects). This also allows recursive operations to collect the set of objects that are potentially changed by a recursive function. Obviously, similar to `@pre` the use of `->modifiedOnly()` is restricted to postconditions.

The definition of the semantics for `->modifiedOnly()` based on the referential universe (see previous section) is straight-forward:

$$X\texttt{->modifiedOnly()} \equiv \lambda(\tau,\tau').\, \lfloor \forall i \notin \left(\text{oidOf}\,{}^{\backprime}\ulcorner X(\tau,\tau')\urcorner\right).\, \tau\, i = \tau'\, i \rfloor$$

where oidOf is just the projection of an object representation to its *oid*. By the projection, the object set X represents a set of references to values in the store. All objects with `oid`'s not occurring in the set are assumed to be unchanged; for `oid`'s occurring in the set, nothing is specified. Thus, requiring `Set{}->modifiedOnly()` in a postcondition of an operation allows for stating explicitly that an operation is a query in the sense of the OCL standard, i. e., the `isQuery` property is true.

Our definition enjoys the property and important proof rule:

$$\tau \vDash S\texttt{->excludes}(x)\ \texttt{and}\ S\texttt{->modifiedOnly()}\ \texttt{implies}\ x = x\texttt{@pre}$$

As can be seen, `x = x@pre` refers to the hole object representation which are therefore attribute-wise equal. Note, further, that the "one-to-one-correspondence between objects and oid's" only holds *within* one state; of course, between different states, this correspondance may change.

Our concept is essentially identical (but simpler) to [Kos06], where query-methods were required to produce no *observable* change of the state (i. e., internally, some objects may have been created, but must be inaccessible at the end). In contrast to frame properties in JML [LBR99], which allow for specifying *attributes* to be assignable or not, our mechanism is a *semantic* and not a *syntactic* one that just forbids assignments to certain paths. This solves the alias problem: `Set{self.a}->modifiedOnly()` is equivalent to `Set{self}->modifiedOnly()` if and only if `self.a` and `self` denote the same object.

6 Conclusion

In our view, there is the need to complement the UML/OCL standardization process by continuous efforts to find a formal semantics. Ideally, this should be a machine-checked semantics like [BW06a] that might become part of the standard document. As can be seen by similar standardization processes (as, for example, the ISO standardization process of the Z language [ISO02]), such a "beau ideal" semantics has the advantage to turn UML into a real formal

method with its potential for high-quality analysis and verification tools. The latter paves the way for light-weight approaches such as [HDWY06] for large-scale industrial applications as well as more heavy-weight system verifications satisfying even EAL7 certification levels ("Formally Verified Design and Tested") of the Common Criteria international standard (ISO/IEC 15408).

It can be safely stated that in contrast to the wealth of informal papers on OCL semantics, a machine-checked semantics results in a higher degree of completeness and perfection. Its main advantage is that it can be used to build an *integrated* semantics, covering data-oriented specification, behavioral specification and programming-language like facets of the UML. Thus, different stakeholders in the standardization process could provide an extension of their proposed UML extension by an extension of the current version of the "beau ideal" semantics to see if their proposed features are in fact consistent with the language. Building such an integrated semantics by paper-and-pencil reasoning or by a design-by-committee process is doomed to failure in the light of our experience.

On the other hand, each attempt to build a formal semantics also results in a certain inflexibility, a lesson that can also be learned from the Z standardization process. This holds to an even larger extent if the semantic representation is machine-checked, requiring that at least representatives of the various stakeholders have sufficient technical skills to handle the underlying theorem prover technology. Similar to standardization efforts centered around a reference implementation, a slow-down of the process is inevitable the more features have been added to the language.

Admittedly, starting the semantics formalization process too early can kill the standardization process as a whole. Not starting it at all, or remaining in a state where only partial approaches exist, will result in a huge inconsistent piece of IT literature. Finding the right balance between informal requirements capture and formalization efforts in the semantics and finding the right point in time to make formal semantics more mandatory in the UML standardization process will therefore be, in our view, crucial for the long-term success of the UML in the future.

References

[And86] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, May 1986.

[BRW03] A. D. Brucker, F. Rittinger, B. Wolff. HOL-Z 2.0: A Proof Environment for Z-Specifications. *Journal of Universal Computer Science* 9(2):152–172, Feb. 2003.

[BW02] A. D. Brucker, B. Wolff. HOL-OCL: Experiences, Consequences and Design Choices. In Jézéquel et al. (eds.), *UML 2002*. LNCS 2460, pp. 196–211. Springer, 2002.

[BW03] A. D. Brucker, B. Wolff. Using Theory Morphisms for Implementing Formal Methods Tools. In Geuvers and Wiedijk (eds.), *Types for Proof and Programs*. LNCS 2646, pp. 59–77. Springer, 2003.

[BW06a] A. D. Brucker, B. Wolff. The HOL-OCL Book. Technical report 525, ETH Zürich, 2006.

[BW06b] A. D. Brucker, B. Wolff. A Package for Extensible Object-Oriented Data Models with an Application to IMP++. In Roychoudhury and Yang (eds.), *SVV 2006*. Computing Research Repository (CoRR). Aug. 2006.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic* 5:56–68, 1940.

[CKM+02] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, A. Wills. The Amsterdam Manifesto on OCL. In Clark and Warmer (eds.), *Object Modeling with the OCL: The Rationale Behind the Object Constraint Language*. LNCS 2263, pp. 115–149. Springer, 2002.

[GJSB00] J. Gosling, B. Joy, G. Steele, G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Prentice-Hall, 2000.

[GM93] M. J. C. Gordon, T. F. Melham. *Introduction to HOL*. Cambridge University Press, July 1993.

[HCH+98] A. Hamie, F. Civello, J. Howse, S. Kent, M. Mitchell. Reflections on the Object Constraint Language. In Bézivin and Muller (eds.), *The Unified Modeling Language. «UML» '98: Beyond the Notation*. LNCS 1618. Springer, June 1998.
doi:10.1007/b72309

[HDWY06] B. Hackett, M. Das, D. Wang, Z. Yang. Modular checking for buffer overflows in the large. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*. Pp. 232–241. ACM Press, 2006.
doi:10.1145/1134285.1134319

[HHB02] R. Hennicker, H. Hussmann, M. Bidoit. On the precise Meaning of OCL Constraints. In Clark and Warmer (eds.), *Object Modeling with the OCL: The Rationale Behind the Object Constraint Language*. LNCS 2263, pp. 69–84. Springer, 2002.

[HR04] D. Harel, B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer* 37(10):64–72, Oct. 2004.
doi:10.1109/MC.2004.172

[Häh91] R. Hähnle. Towards an Efficient Tableau Proof Procedure for Multiple–Valued Logics. In Börger et al. (eds.), *Computer Science Logic (CSL 90)*. LNCS 533, pp. 248–260. Springer, Oct. 1991.
doi:10.1007/3-540-54487-9

[ISO02] ISO/IEC. Information technology – Z formal specification notation – Syntax, type system and semantics. International standard ISO/IEC 13568:2002, International Standards Organisation, July 2002.

[KC86] S. N. Khoshafian, G. P. Copeland. Object identity. In *OOPSLA '86: Conference proceedings on Object-oriented programming systems, languages and applications*. Pp. 406–416. ACM Press, 1986.
doi:10.1145/28697.28739

[Kos06] P. Kosiuczenko. Specification of Invariability in OCL. In Nierstrasz et al. (eds.), *MoDELS 2006: Model Driven Engineering Languages and Systems*. LNCS 4199. Springer-Verlag, Genova, 2006.

[LBR99] G. T. Leavens, A. L. Baker, C. Ruby. JML: A Notation for Detailed Design. In Kilov et al. (eds.), *Behavioral Specifications of Businesses and Systems*. Pp. 175–188. Kluwer Academic Publishers, 1999.

[LW94] B. H. Liskov, J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems* 16(6):1811–1841, Nov. 1994. doi:10.1145/197320.197383

[MC99] L. Mandel, M. V. Cengarle. On the expressive power of OCL. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I*. LNCS 1708. Springer, Sept. 1999.

[OMG97] Object Constraint Language Specification, v1.1. Sept. 1997. Available as OMG document ad/97-08-08.

[OMG99] OMG Unified Modeling Language Specification, v1.3. June 1999. Availalbe as OMG document ad/99-06-08.

[OMG03a] UML 2.0 OCL Specification. Oct. 2003. Available as OMG document ptc/03-10-14.

[OMG03b] OMG Unified Modeling Language Specification, v1.5. Mar. 2003. Available as OMG document formal/03-03-01.

[OMG06] UML 2.0 OCL Specification. May 2006. Available as OMG document formal/06-05-01.

[Ric02] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, 2002.

[RW03] N. Rauch, B. Wolff. Formalizing Java's Two's-Complement Integral Type in Isabelle/HOL. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03)*. ENTCS 80, pp. 1–18. Elsevier Science Publishers, August 2003. doi:10.1016/S1571-0661(04)80808-9

[VJ00] M. Vaziri, D. Jackson. Some Shortcomings of OCL, the Object Constraint Language of UML. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*. P. 555. IEEE Computer Society, 2000. doi:10.1109/TOOLS.2000.10063

[WD96] J. Woodcock, J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.