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) inconsisten-
cies 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 pro-
gram verifications and specification refinement, which are, in our view,
necessary to make OCL more fit for future extensions.

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.

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 deduc-
tive 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 [29,3] or VDM, which, following a
long platonic tradition in logics, start with the notion of *values* and then model
(hierarchies of) relations over them. On the other hand, this remarkably different
perspective makes OCL semantics (and object-oriented specification as a whole)
difficult; numerous luke-warm attempts to integrate object-orientation into spec-
ification formalisms, such as VDM++ or Object-Z, report—among many useful
things—on this particular difficulty. Comparing OCL with the two related ap-
proaches 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 operations in class diagrams.

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 [1,6]. 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 [4], 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 standard's "Semantics" chapter [24, Appendix A]. The question may arise why this original formalization is not adequate for our goals. There are two reasons:

The fundamental reason results from the fact that [24, Appendix A] is based on naive set theory and an informal notion of "model." It assumes a universe for values and objects and algebras over it without any concern of existence and consistency. This paper-and-pencil semantics cannot be strongly formalized in this form, neither in an untyped set theory like Isabelle/ZF or a typed set theory residing in Isabelle/HOL. Since OCL is a typed language at the end, and since we wanted to have type-issues handled by the Isabelle type-checker and not inside the logic representation, it seems more natural to opt for a typed meta-language (like HOL).

The technical reason is a consequence of our design choice to represent the types of OCL expressions one-to-one by HOL types (i.e., the map is injective) such that only well-typed OCL formulae exist in the semantic representation in HOL. Consequently, all well-formedness-related side-conditions are unnecessary in calculi. 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 underlying conservative methodology in more detail. We outline the shallow representation and show its equivalence to [24, Appendix A].

2.1 Higher-order Logic

Higher-order Logic (HOL) [8,2] 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 combination of a typed functional programming language like SML or 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 $[$bool, bool$] \Rightarrow$ bool, $\neg\_$ has type bool $\Rightarrow$ bool. The quantifiers $\forall\_.\_$ and $\exists\_.\_$ have type $[\alpha \Rightarrow$ 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 [10] 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 [24], 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 $\perp$. For all types in this class, we define a combinator strictify by:

$$\text{strictify } f \ x \equiv \text{if } x = \perp \text{ then } \perp \text{ else } f \ x$$

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 $\tau_\perp$ that assigns to each type τ a type *lifted* by $\perp$. Per construction, each type $\tau_\perp$ is in fact in the type class bot. The function $\llcorner\_\lrcorner : \alpha \to \alpha_\perp$ denotes the injection, the function $\ulcorner\_\urcorner : \alpha_\perp \to \alpha$ its inverse for defined values.

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

$\text{lift}_0 \ f \equiv \lambda \tau. \ f$ of type $\alpha \Rightarrow V_\tau(\alpha)$,

$\text{lift}_1 \ f \equiv \lambda X \ \tau. \ f(X \ \tau)$ of type $(\alpha \Rightarrow \beta) \Rightarrow V_\tau(\alpha) \Rightarrow V_\tau(\beta)$, and

$\text{lift}_2 \ f \equiv \lambda X \ Y \ \tau. \ f(X \ \tau)(Y \ \tau)$ of type $([\alpha, \beta] \Rightarrow \gamma) \Rightarrow [V_\tau(\alpha), V_\tau(\beta)] \Rightarrow V_\tau(\gamma)$.

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 [5]. 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:

$$1 \equiv \text{lift}_0(\llcorner 1 \lrcorner)$$
$$\text{not } \_ \equiv \text{lift}_1(\text{strictify}(\llcorner\_\lrcorner \circ (\neg\_) \circ \ulcorner\_\urcorner)$$
$$\_ + \_ \equiv \text{lift}_2(\text{strictify}(\lambda x. \ \text{strictify}(\lambda y. \ \llcorner \ulcorner x \urcorner + \ulcorner y \urcorner \lrcorner)))$$

where $\_\circ\_$ denotes function composition. We use overloading here: the $\_+\_$ on the left-hand side of the last definition has type $[V_\tau(\mathrm{int}_\perp), V_\tau(\mathrm{int}_\perp)] \Rightarrow V_\tau(\mathrm{int}_\perp)$ (where $V_\tau(\mathrm{int}_\perp)$ is the HOL equivalent to the OCL type `Integer`), while the $\_+\_$ on the right-hand-side has type $[\mathrm{int}, \mathrm{int}] \Rightarrow \mathrm{int}$. 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, [24, page A-11] presents this definition as:

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

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. $I[\![w(e_1, \ldots e_n)]\!]\tau = I(w)(\tau)\big(I[\![e_1]\!](\tau), \ldots, I[\![e_n]\!](\tau)\big)$

<div align="right">(OCL Specification [24], page A-26, definition A.30)</div>

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* [24, 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 re-introduce a kind of "explicit semantic function" $I[\![E]\!]\tau$ into our shallow embedding as a syntactic marker, i.e., by stating the identity:

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

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:

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

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 [24, 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 {true, false} ∪ {⊥} used in the standard are simply illegal in our interpretation. This makes the injections $\lfloor \_ \rfloor$ and projections $\lceil \_ \rceil$ necessary.

2. The semantic functions in the standard are split into $I(x)$, $I[\![e]\!]\tau$, $I_{\text{ATT}}[\![e]\!]\tau$ 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 [6].

2.4 The Benefits of a "Strong" Formal Semantics

Our strong formalization of [24, 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 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 [24, Appendix A]. We provide formal proofs for the compliance of our OCL semantics with these requirements (see [6] 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

OMG standards are developed in an open process by the OMG (Object Management Group) leading to a variety of (intermediate) "standardization" documents. Especially for UML and OCL, which have a long history. OCL was introduced as an OMG specification language as additional document [22] completing the UML 1.1 standard [23]. 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., [25, 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 [28,21,14,9] in academia and this discussions clearly fertilized the development towards OCL 2.0. Especially the work of Richters [27] 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 *informative*, i.e., not normative, parts. The semantics of the standard appears in the following chapters of [24]:

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 under-specified "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 [27] , 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.

---

[1] A good overview of the different usages of the word "semantics" is given in [15].

Nevertheless, we think the OCL standard [24] ("`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 ("`ptc/06-05-01`"), are an ad-hoc attempt to align OCL 2.0 to the UML 2.0 and represent a considerable step back with respect to consistency and potential for formal semantics. Nevertheless, all issues addressed in this paper are also valid for "`ptc/06-05-01`".

In the remainder of this section, we will explain some selected problems; our choice focuses on semantical problems which, among others, are caused by inconsistencies or missing concepts in the standard document.

4.2 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. [24, Chapter 11] requires the following specification of implies:

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

2. [24, Appendix A] defines the `b1 implies b2` by a truth table:

b1 \ b2	false	true	⊥
false	true	true	true
true	false	true	⊥
⊥	⊥	true[‡]	⊥

While we were checking the consistency of the formal semantics [24, Chapter A] with the normative requirements [24, Chapter 11], we detected an inconsistency: calculating the truth table for the definitions of `implies` given in the normative part one would expect ⊥ instead of `true` on the position marked with an [‡]. This inconsistency could be changed either by changing the truth tables [24, Chapter A] or by changing the requirements [24, Chapter 11] to:

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

which represents the "classical definition" of implication.

Whereas different variants for implications for three-valued logics are considered in the literature [13,16], 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":

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

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, 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 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. Nevertheless, there are two important exceptions to this rule: the logical connectives and the collection constructors. Whereas for the logical connectives this exception is stated both in the normative part [24, Chapter 11] and in the informative part [24, Appendix A], for the collection constructors this is only explained in the informative part [24, Appendix A]. The normative part of the standard does not cover this issue.

In the literature, sets with strict constructors are called *smashed*. Such smashed set types often occur in semantics for programming languages, e.g., SML. In a language with semantic domains providing $\bot$-elements, the question arises how they are treated in type constructors like product, sum, list or sets. Two extremes are known in the literature; for products, for example, we can have:

$$(\bot, X) \neq \bot \qquad\qquad \{a, \bot, b\} \neq \bot \qquad\qquad \ldots$$

or:

$$(\bot, X) = \bot \qquad\qquad \{a, \bot, b\} = \bot \qquad\qquad \ldots$$

The latter variant is called *smashed product* and *smashed set*. The normative chapters make no clear decision here. We strongly opt for a smashed collection semantics, based on two reasons:

1. OCL tends to define its constructs towards executablility and proximity to object-oriented programming languages such as Java, and more important
2. OCL with non-smashed collection semantics leads to very complicated logical calculi. Just consider the rule

$$\frac{\texttt{self.OclIsDefined()}}{\texttt{self->forAll(e | e.OclIsDefined())}}$$

which only holds for a smashed semantics. Without such rules, reasoning over navigations, i.e., collections, always requires a proof of the definedness of all elements of a navigation.

To study the effects of a non-smashed collection semantics on formal reasoning, we provide a separate configuration of HOL-OCL, details can be found in [6].

4.4 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 [25, chapter 4.4.1] requires that operation names are unique within the same namespace. In particular, subclasses may not over-write inherited operations. Albeit, the UML standard allows one to (explic-itly) overwrite *methods*, i.e., *implementations* of operations.
2. The OCL standard [24, 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 precon-ditions 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 opera-tion (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. Nev-ertheless, 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 [6], 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 con-servativity will impose restrictions on invocations here to be discussed in sub-section 4.6.

4.5 Equalities

Historically, object-oriented systems are equipped with a variety of different "equalities" [18]. Answering the question whether two objects are equal is not so obvious. For example, are two objects equal only if their object identifiers are equal (are they the same object?) or are two objects equal if their values are equal? Whereas in traditional specification formalisms the equality is defined over values, the most basic equality over objects is the reference equality or identity equality, which is also the kind of equality that is usually provided as a default, i.e., "built-in," equality in object-oriented programming languages. Thus there is a fundamental difference between values and objects.

This situation, i.e., which role do references play within OCL, is not clearly stated in the OCL standard. ([6] gives a detailed discussion of this topic) and we will only discuss in this paper the consequences of taking undefinedness, e.g., values and references can be undefined, into account. Further, the well-known equivalence properties need to be generalized, e.g., *symmetry* ($x = y \Rightarrow y = x$) is generalized to *quasi-symmetry* ($x = y \Rightarrow y = x$ for x and y being defined).

Naturally, we can apply the concept of strictness to an equality operator: an equality operator is called *strict equality* if it evaluates to undefined whenever one of its arguments is undefined, i.e., if the following properties hold:

$$(o \doteq \bot) = \bot \,, \qquad (\bot \doteq o) = \bot \,, \text{ and} \qquad (\bot \doteq \bot) = \bot \,.$$

In contrast, an equality operator is called a *strong equality* if it satisfies the property: $(\bot \triangleq \bot) = \mathsf{T}$.

The OCL standard defines equality as the strict equality over values [24, Sec. A.2.2], and since objects are values, and object identifiers are not distinguished from object values [24, Definition A.10] we chose the strict equality $\_ \doteq \_$ as the default OCL equality within HOL-OCL. Nevertheless, several interesting properties, like being quasi-reflexive, quasi-symmetric, quasi-transitive and quasi-substitutive only hold for the strong equality (even potentially undefined values can be substituted). Therefore, the strong equality is of outstanding importance for deduction.

Also, consider for example 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. Thus we suggest to extend OCL with a strong equality operation.

4.6   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 [24], 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 [24], 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 [20] 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 converted into a format that satisfies the constraints of 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 the case-switch:

$$\text{if } self \rightarrow \texttt{IsType}(A) \text{ then } S \text{ else if } \dots \text{ else } S''$$

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

In more recent draft-versions of the standard, *bounded* versions of Integers were suggested. If the purpose of these types is to allow a transition to implementation

languages, then we suggest to choose also a concrete machine arithmetic based on a two's complement model (as described in [11]; similar concrete definitions for C++ are in preparation). These machine arithmetics are somewhat more distant to mathematics (MaxInt $+ 1 =$ MinInt, all commutative ring properties hold except $a - a = 0$, $0 \leq \mathrm{abs}(a)$ does not hold even for defined a, etc.) but close to widely used implicit standards in microprocessor technology. Verification of such transitions from mathematical integers to machine integers can be a real concern in safety-critical applications. For a concrete proposal of a "strong" formalization of the Java arithmetics, see [26].

It is conceivable to drop the standards limitation to *finite* collection types. Infinite sets are clearly a very powerful and useful standard 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.

5.2   References and Referential Types

One way to view objects with attributes $a_1, \ldots a_n$ of OCL types $T_1, \ldots, T_n$ are tuples of type $\tau_1 \times \ldots \times \tau_n$ where τ_i is the corresponding HOL type of $T_i{}^2$. An object universe can therefore be constructed as the sum over all Cartesian products representing objects. The state of a system is then a partial map from object-id's (*oid*'s) to the object universe.

We suggest one little change of this scheme: the oid should be encoded into the Cartesian products as well, comparable to a "hidden attribute": $\mathrm{oid} \times \tau_1 \times \ldots \times \tau_n$, and states should be restricted to contain only objects whose oid points to itself in the state. This invariant is easy to implement by constructors in an object-oriented programming language: just generate the fresh oid and store it in the object. However, this slight change makes the logical equality $\_ \triangleq \_$ coincide with the referential equality used in many programming languages—as long as we are comparing objects within one state. This makes the formal model of sets, for example, which implicitly uses this equality, much more realistic. This extension of the object encoding scheme is called *referential (object) universe* in [6].

Moreover, this extension also has advantages for programming languages used for implementing methods (see IMP++ for example in [7], which also contains a verification technique). For example, having a reference attribute in each object greatly simplifies the semantic definition of a reference operator (like e.g., `&x` in C++, mapping the type of `x` to its reference type). Further, the assignment operator assigning a reference to an attribute of type oid can then also

---

² We ignore the complications resulting from extensible subtyping here; see [6] for details

be realized easily. Referential universes also give a possible interpretation of the "null-objects" mentioned in the recent versions of the standard "`ptc/06-05-01`".

5.3 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:

$$X\text{->modifiedOnly}() \equiv \lambda(\tau,\tau').\ \lfloor \forall i \notin \big(\text{oidOf} \ ` \ulcorner X(\tau,\tau')\urcorner\big).\ \tau\ i = \tau'\ i \rfloor$$

where oidOf is just the projection of an object 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.

In contrast to frame properties in JML [19], 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 just equivalent to `Set{self}->modifiedOnly()` iff `self.a` and `Set{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 [6] 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 [17]), 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 [12] 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.

Acknowledgement:

References

1. The HOL-OCL website, 2006. `http://www.brucker.ch/projects/hol-ocl/`.
2. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.
3. A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, 2003.
4. A. D. Brucker and B. Wolff. HOL-OCL: Experiences, consequences and design choices. In J.-M. Jézéquel, H. Hussmann, and S. Cook, eds., *UML 2002*, no. 2460 in LNCS, pp. 196–211. Springer, 2002.

5. A. D. Brucker and B. Wolff. Using theory morphisms for implementing formal methods tools. In H. Geuvers and F. Wiedijk, eds., *Types for Proof and Programs*, no. 2646 in LNCS, pp. 59–77. Springer, 2003.

6. A. D. Brucker and B. Wolff. The HOL-OCL book. Tech. Rep. 525, ETH Zürich, 2006.

7. A. D. Brucker and B. Wolff. A package for extensible object-oriented data models with an application to IMP++. In A. Roychoudhury and Z. Yang, eds., *SVV 2006*, Computing Research Repository (CoRR). 2006.

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

9. S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The Amsterdam Manifesto on OCL, 1999.

10. M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

11. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, 2000.

12. B. Hackett, M. Das, D. Wang, and 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.

13. R. Hähnle. Towards an efficient tableau proof procedure for multiple–valued logics. In E. Boerger, H. K. Buening, M. M. Richter, and W. Schoenefeld, eds., *CSL 90*, LNCS, vol. 533, pp. 248–260. Springer, 1991.

14. A. Hamie, F. Civello, J. Howse, S. Kent, and M. Mitchell. Reflections on the Object Constraint Language. In *Post Workshop Proceedings of UML 98*. Springer, 1998.

15. D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, pp. 64–72, 2004.

16. R. Hennicker, H. Hussmann, and M. Bidoit. On the precise meaning of OCL constraints. In T. Clark and J.Warmer, eds., *Advances in Object Modelling with the OCL*, LNCS, vol. 2263, pp. 69–84. Springer, 2002.

17. Formal Specification – Z Notation – Syntax, Type and Semantics. 2002. Draft International Standard.

18. S. N. Khoshafian and G. P. Copeland. Object identity. In *OOPSLA 86*, pp. 406–416. ACM Press, 1986.

19. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, eds., *Behavioral Specifications of Businesses and Systems*, pp. 175–188. Kluwer Academic Publishers, 1999.

20. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. 16(6):1811–1841, 1994.

21. L. Mandel and M. V. Cengarle. On the expressive power of OCL. In *FM 99*. 1999.

22. Object constraint language specification. 1997. Covers OCL 1.1.

23. OMG Unified Modeling Language Specification. 1999. Covers UML/OCL 1.3.

24. UML 2.0 OCL specification. 2003. `ptc/2003-10-14`.

25. OMG Unified Modeling Language Specification. 2003. `formal/03-03-01`.

26. N. Rauch and B. Wolff. Formalizing Java's two's-complement integral type in isabelle/HOL. In *ENTCS*, vol. 80. Elsevier Science Publishers, 2003.

27. M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. Ph.D. thesis, Universität Bremen, 2002.

28. M. Vaziri and D. Jackson. Some shortcomings of OCL, 1999. Response to Object Management Group's Request for Information on UML 2.0.

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