Extending OCL with Null-References

Towards a Formal Semantics for OCL 2.1

Achim D. Brucker[1], Matthias P. Krieger[2], and Burkhart Wolff[2]

[1] SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com
[2] Université Paris-Sud, Parc Club Orsay Université, 91893 Orsay Cedex, France
{krieger, wolff}@lri.fr

Abstract From its beginnings, OCL is based on a strict semantics for undefinedness, with the exception of the logical connectives of type Boolean that constitute a three-valued propositional logic. Recent versions of the OCL standard added a second exception element, which, similar to the null references in object-oriented programming languages, is given a non-strict semantics. Unfortunately, this extension has been done in an ad hoc manner, which results in several inconsistencies and contradictions. In this paper, we present a consistent formal semantics (based on our HOL-OCL approach) that includes such a non-strict exception element. We discuss the possible consequences concerning class diagram semantics as well as deduction rules. The benefits of our approach for the specification-pragmatics of design level operation contracts are demonstrated with a small case-study.
Key words: HOL-OCL, UML, OCL, null reference, formal semantics

1 Introduction

The Object Constraint Language (OCL) has established itself as a language for annotating models with constraints that are not expressed graphically. OCL is used for specifying model constraints such as well-formedness rules and for defining object-oriented designs through operation contracts and class invariants. The expressions of OCL constitute the core of the language. In essence, OCL allows for evaluating queries over UML models.

From its beginnings, OCL has been equipped with the notion of an undefined value called `invalid` to deal with exceptions occurring during expression evaluation. A classical example of such an exception is a division by zero. In OCL such an erroneous division is specified to yield an undefined value. Other reasons for exceptions include attempting to retrieve elements from empty collections, illegal type conversions and evaluating attributes on objects that do not exist. Most operations in OCL are defined to be *strict*, i. e., they evaluate to `invalid` if they are called with an undefined argument. This ensures that errors are propagated during expression evaluation so they are visible and handled later on. Naturally, OCL collections are not allowed to have undefined elements, since errors are more easily signaled by marking the collection value as undefined.

During the development of OCL, the potential benefits of a second exception element in addition to `invalid` became clear. The desired second exception element, called `null`, is intended to represent the *absence of value* rather than to indicate evaluation errors. The need to express the absence of value arises naturally when dealing with object attributes with a multiplicity lower bound of zero. These attributes, that occur frequently in models, are not required to yield a value when evaluated. Representing this absence of value with the original undefined value `invalid` would be inconvenient and counter intuitive. To prevent a propagation of undefined values, it would be necessary to handle all cases of value absence immediately. In particular, it would not be possible to pass potentially null values to strict operations. Since nearly all operations of OCL are strict, even the most basic operations such as equality testing would not be realizable without checking for an absence of value. These difficulties can be avoided by introducing the second exception element `null` as a valid operation argument and collection element.

The latest OCL 2.0 standard [17] introduces `null` as a second exception element representing the absence of a value. Unfortunately, this extension has been done in an ad hoc manner, which results in several inconsistencies and contradictions. For example, both `invalid` and `null` are defined to conform to all classifiers, in particular `null` conforms to `invalid` and vice versa. Since the conforms relationship is antisymmetric, this implies that `invalid` and `null` are indistinguishable. The standard does not make clear when object attributes can evaluate to `null` values and how this depends on the multiplicity of the attribute. There is also no indication in the standard whether objects that do not exist ("dangling references") are treated the same way as `null` or not. Unsurprisingly, a recent evaluation [11] of OCL tools identified the handling of undefined values as a major weakness of most tools.

To overcome these problems in the current version of the OCL standard, we present a consistent formal semantics (based on our HOL-OCL approach [6, 4]) that includes null as non-strict exception element. The paper is organized as follows. In Section 3, we provide a textbook-style summary of the essentials of the HOL-OCL semantics, which is a strong formal, i. e., machine-checked semantics largely following [16, Annex A]. In Section 4, we present as an increment our proposal for OCL 2.1, focusing to the key issue of null-elements and null-types. In Section 5, we will discuss the consequences for an omnipresent feature of UML, namely multiplicities, and its pragmatics. Finally, in Section 6 we will show how the extended language can be used to describe pretty standard contracts at design-level for object-oriented programs.

2 Formal and Technical Background

2.1 Higher-order Logic

Higher-order Logic (HOL) [8, 1] 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 $\alpha :: \mathit{order}$ or $\alpha :: \mathrm{bot}$) or *type constructors* (like bool or nat). Type constructors may have arguments (as in $\alpha\,\mathrm{list}$ or $\alpha\,\mathrm{set}$). 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 $[\alpha, \alpha] \Rightarrow \mathrm{bool}$, 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 $\_ \land \_$, $\_ \lor \_$, $\_ \Rightarrow \_$ of HOL have type $[\mathrm{bool}, \mathrm{bool}] \Rightarrow \mathrm{bool}$, $\neg\_$ has type $\mathrm{bool} \Rightarrow \mathrm{bool}$. The quantifiers $\forall\_.\_$ and $\exists\_.\_$ have type $[\alpha \Rightarrow \mathrm{bool}] \Rightarrow \mathrm{bool}$. The quantifiers may range over types of higher order, i. e., functions or sets. Sets of type $\alpha\,\mathrm{set}$ can be defined isomorphic to functions of type $\alpha \Rightarrow \mathrm{bool}$; the definition of the element-hood $\_ \in \_$, the set comprehension $\{\_.\_\}$, as well as $\_ \cup \_$ and $\_ \cap \_$ are standard.

The entire Isabelle/HOL library, including typed set theory, well-founded recursion theory, number theory and theories for data-structures like Cartesian products $\alpha \times \beta$ and disjoint type sums $\alpha + \beta$ is built on top of the HOL core-language. The library also includes the type constructor $\tau_\bot$ that assigns to each type τ a type *lifted* by the exceptional element $\bot$. The function $\llcorner\_\lrcorner : \alpha \Rightarrow \alpha_\bot$ denotes the injection, the function $\ulcorner\_\urcorner : \alpha_\bot \Rightarrow \alpha$ its inverse for defined values. On this basis, partial functions $\alpha \rightharpoonup \beta$ are just defined as functions $\alpha \Rightarrow \beta_\bot$ over which the usual concepts of domain $\mathrm{dom}\ f$ and range $\mathrm{ran}\ f$ are introduced. The library is built entirely by logically safe, conservative definitions and derived rules. This methodology is also applied to HOL-OCL.

2.2 A Brief Introduction to the HOL-OCL System

HOL-OCL [6, 4] is integrated into a framework [3] supporting a formal, model-driven software engineering process. Technically, HOL-OCL is based on a repository for UML/OCL models, called su4sml, and on Isabelle/HOL; both are written in SML. HOL-OCL also reuses and extends the existing Isabelle front-end called Proof General as well as the Isabelle documentation generator. Figure 1 gives an architecture overview of the main system components of HOL-OCL, namely:

- the data repository, called su4sml, providing XMI import facilities,
- the datatype package, or *encoder*, which encodes UML/OCL models into HOL; from a user's perspective, it yields a semantic interface to the model,
- the HOL-OCL library which provides the definitions of the semantics discussed here and the derived core theorems needed for verification, and
- a suite of proof procedures based on rewriting and tableaux techniques.

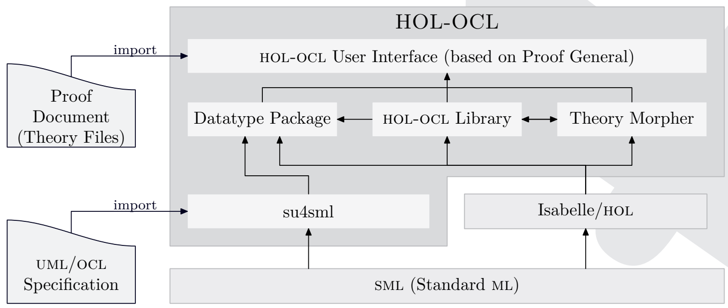

Figure 1. Overview of the HOL-OCL architecture.

3 An Overview over OCL Semantics

In this section, we will briefly introduce to OCL semantics from the HOL-OCL perspective. The main differences between the OCL 2.0 formal semantics description [16, Annex A] and HOL-OCL is 1) that the latter is a machine-checked, "strong" formal semantics which is itself based 2) on a typed meta-language (i. e., HOL) instead of an untyped one (i. e., naïve set theory), and 3) various technical simplifications: instead of three different semantic interpretation functions $I(x)$, $I[\![e]\!]\tau$, $I_{\mathrm{ATT}}[\![e]\!]\tau$, we use only one. The first difference enables us to give a semantic consistency guarantee: Since all definitions of our formal semantics are logically safe extensions, i. e., *conservative* [12] and since all rules are derived, the consistency of HOL-OCL is reduced to the consistency of HOL, i. e., a widely accepted small system of seven axioms. The second difference dramatically reduces the number of rules necessary for formal reasoning.

In this presentation we will avoid to show the key-definitions used inside HOL-OCL; rather, for the sake of making this work amenable to a wider audience, we will use a "textbook-style" presentation of the semantics which is formally shown to be equivalent (see also [6]).

3.1 Validity and evaluations

The topmost goal of the formal semantics is to define the *validity statement*:

$$(\sigma, \sigma') \vDash P\,,$$

where σ is the pre-state and σ' the post-state of the underlying system and P is a Boolean expression (a *formula*). The assertion language of P is composed of
1. operators on built-in data structures such as Boolean or set,
2. operators of the user-defined data-model such as accessors, type-casts and tests, and
3. user-defined, side-effect-free methods.

Informally, a formula P is valid if and only if its evaluation in the context (σ, σ') yields true. As all types in HOL-OCL are extended by the special element $\perp$

denoting undefinedness, we define formally:

$$(\sigma, \sigma') \vDash P \equiv \big(P(\sigma, \sigma') = {}_\llcorner\mathrm{true}_\lrcorner\big)\,.$$

Since all operators of the assertion language depend on the context (σ, σ') and result in values that can be $\bot$, all expressions can be viewed as *evaluations* from (σ, σ') to a type $\tau_\bot$. Consequently, all types of expressions have a form captured by the following type abbreviation

$$V(\alpha) := \mathrm{state} \times \mathrm{state} \Rightarrow \alpha_\bot\,,$$

where state stands for the system state and state × state describes the pair of pre-state and post-state and _ := _ denotes the type abbreviation.

The OCL semantics [16, Annex A] uses different interpretation functions for invariants and pre-conditions; we achieve their semantic effect by a syntactic transformation $\__{\mathrm{pre}}$ which replaces all accessor functions _. a by their counterparts _. a `@pre`. For example, $(\mathit{self}.\,\mathrm{a} > 5)_{\mathrm{pre}}$ is just $(\mathit{self}.\,\mathrm{a}\,\texttt{@pre} > 5)$.

3.2 Strict operations

Following common terminology, an operation that returns $\bot$ if one of its arguments is $\bot$ is called *strict*. The majority of the operations is strict, e. g., the Boolean negation is formally presented as:

$$I[\![\texttt{not}\ X]\!]\tau \equiv \begin{cases} {}_\llcorner\ulcorner\neg I[\![X]\!]\tau\urcorner{}_\lrcorner & \text{if } [\![X]\!]\tau \neq \bot,\\ \bot & \text{otherwise}\,. \end{cases}$$

where $\tau = (\sigma, \sigma')$ and $I[\![\_]\!]$ is a notation marking the HOL-OCL constructs to be defined. This notation is motivated by the definitions in the OCL standard [16]. In our case, $I[\![\_]\!]$ is just the identity, i. e., $I[\![X]\!] \equiv X$. These operators can be viewed as *transformers on evaluations*; the HOL type of `not` _ is $V(\mathrm{bool}) \Rightarrow V(\mathrm{bool})$.

The binary case of the integer addition is analogous:

$$I[\![X + Y]\!]\,\tau \equiv \begin{cases} {}_\llcorner\ulcorner I[\![X]\!]\,\tau\urcorner + \ulcorner I[\![Y]\!]\,\tau\urcorner{}_\lrcorner & \text{if } I[\![X]\!]\,\tau \neq \bot \text{ and } I[\![Y]\!]\,\tau \neq \bot,\\ \bot & \text{otherwise}\,. \end{cases}$$

Here, the operator _ + _ on the right refers to the integer HOL operation with type $[\mathrm{int}, \mathrm{int}] \Rightarrow \mathrm{int}$. The type of the corresponding strict HOL-OCL operator _ + _ is $[V(\mathrm{int}), V(\mathrm{int})] \Rightarrow V(\mathrm{int})$.

A slight variation of this definition scheme is used for the operators on collection types such as HOL-OCL sets, sequences or bags:

$$I[\![X\texttt{->union}(Y)]\!]\tau \equiv \begin{cases} S_\llcorner\ulcorner I[\![X]\!]\tau\urcorner \cup \ulcorner I[\![Y]\!]\tau\urcorner{}_\lrcorner & \text{if } I[\![X]\!]\tau \neq \bot \text{ and } I[\![Y]\!]\tau \neq \bot,\\ \bot & \text{otherwise.} \end{cases}$$

Here, S ("smash") is a function that maps a lifted set ${}_\llcorner X_\lrcorner$ to $\bot$ if and only if $\bot \in X$ and to the identity otherwise. Smashedness of collection types is the natural extension of the strictness principle for data structures.

Intuitively, the type expression $V(\tau)$ is a representation of the type that corresponds to the HOL-OCL type τ. We introduce the following type abbreviations:

$$\texttt{Boolean} := V(\text{bool})\,, \qquad \texttt{Set}(\alpha) := V(\alpha\ \text{set})\,,$$
$$\texttt{Integer} := V(\text{int})\,,\ \text{and} \qquad \texttt{Sequence}(\alpha) := V(\alpha\ \text{list})\,.$$

The mapping of an expression $\texttt{E}$ of HOL-OCL type $\texttt{T}$ to a HOL expression E of HOL type T is injective and preserves well-typedness.

3.3 Boolean Operators

There is a small number of explicitly stated exceptions from the general rule that HOL-OCL operators are strict: the strong equality, the definedness operator and the logical connectives. As a prerequisite, we define the logical constants for truth, absurdity and undefinedness. We write these definitions as follows:

$$I[\![\texttt{true}]\!]\tau \equiv \lfloor\text{true}\rfloor\,, \qquad I[\![\texttt{false}]\!]\tau \equiv \lfloor\text{false}\rfloor\,,\ \text{and} \qquad I[\![\texttt{invalid}]\!]\tau \equiv \bot\,.$$

HOL-OCL has a *strict equality* $\_ \doteq \_$. On the primitive types, it is defined similarly to the integer addition; the case for objects is discussed later. For logical purposes, we introduce also a *strong equality* $\_ \triangleq \_$ which is defined as follows:

$$I[\![X \triangleq Y]\!]\,\tau \equiv (I[\![X]\!]\,\tau = I[\![Y]\!]\,\tau)\,,$$

where the $\_ = \_$ operator on the right denotes the logical equality of HOL. The undefinedness test is defined by $X\,\texttt{.IsDefined()} \equiv \texttt{not}\,(X \triangleq \texttt{invalid})$. The strong equality can be used to state reduction rules like: $\tau \vDash (\texttt{invalid} \doteq X) \triangleq \texttt{invalid}$. The OCL standard requires a Strong Kleene Logic. In particular:

$$I[\![X\ \texttt{and}\ Y]\!]\tau \equiv \begin{cases} \lfloor\ulcorner x\urcorner \wedge \ulcorner y\urcorner\rfloor & \text{if } x \neq \bot \text{ and } y \neq \bot, \\ \lfloor\text{false}\rfloor & \text{if } x = \lfloor\text{false}\rfloor \text{ or } y = \lfloor\text{false}\rfloor, \\ \bot & \text{otherwise}\,. \end{cases}$$

where $x = I[\![X]\!]\tau$ and $y = I[\![Y]\!]\tau$. The other Boolean connectives were just shortcuts: $X\ \texttt{or}\ \ Y \equiv \texttt{not}\,(\texttt{not}\ X\ \texttt{and}\,\texttt{not}\ Y)$ and $X\ \texttt{implies}\ Y \equiv \texttt{not}\ X\ \texttt{or}\ \ Y$. The logical quantifiers are viewed as special operations on the collection types $\texttt{Set}(\alpha)$ or $\texttt{Sequence}(\alpha)$. Their definition in the OCL standard is very operational and restricted to the finite case; instead, we define the universal quantification as generalization of the conjunction:

$$I[\![X\texttt{->forall}(x|P(x)]\!]\tau \equiv$$
$$\begin{cases} \bot & \text{if } I[\![X]\!]\tau = \bot, \\ \lfloor\forall x \in \ulcorner I[\![X]\!]\tau\urcorner.\ \ulcorner I[\![P(\lambda\,\tau.\,x)]\!]\tau\urcorner\rfloor & \text{if } \forall x \in \ulcorner I[\![X]\!]\tau\urcorner.\ I[\![P(\lambda\,\tau.\,x)]\!]\tau \neq \bot, \\ \lfloor\text{false}\rfloor & \text{if } \exists x \in \ulcorner I[\![X]\!]\tau\urcorner.\ I[\![P(\lambda\,\tau.\,x)]\!]\tau = \lfloor\text{false}\rfloor, \\ \bot & \text{otherwise.} \end{cases}$$

The existential quantification $X\texttt{->exists}(x|P(x))$ is defined as the usual abbreviation: $X\texttt{->exists}(x|P(x)) \equiv \texttt{not}\ \ X\texttt{->forall}(x|\,\texttt{not}\ P(x))$.

3.4 Object-oriented Data Structures

In the previous sections, we described various built-in operations on datatypes and the logic. Now we turn to several families of operations that the user implicitly defines when stating a class model as logical context of a specification. This is the part of the language where object-oriented features such as type casts, accessor functions, and tests for dynamic types come into play. Syntactically, a class model provides a collection of classes C, an inheritance relation $\_ < \_$ on classes and a collection of attributes A associated to classes. Semantically, a class model means a collection of accessor functions (denoted $\_.\mathrm{a} :: A \to B$ and $\_.\mathrm{a}\,\texttt{@pre} :: A \to B$ for $\mathrm{a} \in A$ and $A, B \in C$), type casts that can change the static type of an object of a class (denoted $\_\texttt{.oclAsType}(C)$ of type $A \to C$) and dynamic type tests (denoted $\_\texttt{.oclIsTypeOf}(C)$). A precise formal definition of the syntactic side of a class system can be found in [6].

Class models: A simplified semantics In this section, we will have to clarify the notions of *object identifiers*, *object representations*, *class types* and *state*. We will give a formal model for this, that will satisfy all properties discussed in the subsequent section except one; the reader interested in a complete model is referred to (for details, see [5]).

First, object identifiers are captured by just an abstract type oid comprising countably many elements and a special element `nullid`.

Second, object representations model "a piece of typed memory," i. e. a kind of record comprising some administration information and the information for all the attributes of an object; here, the basic types $\texttt{Boolean}_\tau$, $\texttt{Integer}_\tau$, etc. as well as collections over them are stored directly in the object representations, class types and collections over them are represented by oid's (respectively lifted collections over them; such collections may be $\perp$).

Third, the class type C will be the type of such an object representation. It is a Cartesian product:

$$C := (\mathrm{oid} \times \mathrm{C_t} \times A_1 \times \cdots \times A_k)$$

where a unique tag-type $\mathrm{C_t}$ (ensuring type-safety) is created for each class type, and where the types $A_1, \ldots, A_k$ are the attribute types (including all inherited attributes) with class types substituted by type oid. The function OidOf projects the first component, the oid, out of an object representation.

For a class model M with the classes $C_1, \ldots, C_n$, we define states as partial functions from oid's to object representations satisfying a *state invariant* inv_σ:

$$\mathrm{state} := \{f :: \mathrm{oid} \rightharpoonup (C_1 + \ldots + C_n) \mid \mathrm{inv}_\sigma(f)\}$$

where $\mathrm{inv}_\sigma(f)$ states two conditions:

1. there is no object representation for `nullid`: $\texttt{nullid} \notin (\mathrm{dom}\ f)$.
2. there is a"one-to-one" correspondence between object representations and oid's: $\forall \mathit{oid} \in \mathrm{dom}\ f.\ \mathit{oid} = \mathrm{OidOf}\ \ulcorner f(\mathit{oid}) \urcorner$

The latter condition is also mentioned in [16, Annex A] and goes back to Mark Richters [19].

3.5 The Accessors

On states built over object universes, we can now define accessors, casts, and type tests of an object model. We consider the case of an attribute a of class C which has the simple class type D (not a basic type, not a collection):

$$I[\![\mathit{self}.\,\mathrm{a}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } I[\![\mathit{self}]\!](\sigma,\sigma') = \bot \lor \mathrm{OidOf}\,\ulcorner O \urcorner \notin \mathrm{dom}\,\sigma' \\ \mathrm{get}_{\mathrm{D}}\, u & \text{if } \sigma'(\mathrm{get}_{\mathrm{C}}\ulcorner \sigma'(\mathrm{OidOf}\,\ulcorner O \urcorner)\urcorner.\,\mathrm{a}^{(0)}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise.} \end{cases}$$

$$I[\![\mathit{self}.\,\mathrm{a@pre}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } I[\![\mathit{self}]\!](\sigma,\sigma') = \bot \lor \mathrm{OidOf}\,\ulcorner O \urcorner \notin \mathrm{dom}\,\sigma \\ \mathrm{get}_{\mathrm{D}}\, u & \text{if } \sigma(\mathrm{get}_{\mathrm{C}}\ulcorner \sigma(\mathrm{OidOf}\,\ulcorner O \urcorner)\urcorner.\,\mathrm{a}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise.} \end{cases}$$

Here, $\mathrm{get}_{\mathrm{D}}$ is the projection function from the object universe to $D_{\bot}$, and $x.\,\mathrm{a}$ is the projection of the attribute from the class type (the Cartesian product). In the case of simple class type, we have to evaluate expression *self*, get an object representation (or undefined if the evaluation is not possible), project the attribute, de-reference it in the pre or post state, respectively, and project the class object from the object universe ($\mathrm{get}_{\mathrm{D}}$ may yield $\bot$ if the element in the universe does not correspond to a D object representation.) In the case for a basic type attribute, the de-referentiation step is left out, and in the case of a collection over class types, the elements of the collection have to be point-wise de-referenced and smashed.

In our model accessors always yield (type-safe) object representations; not oid's. This has the consequence that a reference, that is *not* in dom σ, i. e., that is a "dangling reference," immediately results in `invalid` (this is a subtle difference to [16, Annex A] where the undefinedness is detected one de-referentiation step later). The strict equality $\_ \doteq \_$ must be defined via OidOf when applied to objects. It satisfies $(\texttt{invalid} \doteq X) \triangleq \texttt{invalid}$.

The definitions of casts and type tests can be found in [5], together with other details of the construction above and its automation in HOL-OCL.

4 A Proposal for an OCL 2.1 Semantics

In this section, we describe our OCL 2.1 semantics proposal as an increment to the OCL 2.0 semantics (currently underlying HOL-OCL and essentially formalizing [16, Annex A]). In later versions of the standard [17] the formal semantics appendix textually reappears although being inherently incompatible with the mandatory parts of the standard.

4.1 Revised Operations on Basic Types

In the UML standard, and since [17] also in the OCL standard, all basic types comprise also the `null`-element, modeling the possibility to be non-existent. Seen from a functional language perspective, this corresponds to the view that each basic value is a type like `int option` as in SML. Technically, this means that any basic type is doubly lifted:

$$\texttt{Integer} := V(\text{int}_{\bot}) \,, \text{ etc.}$$

and basic operations have to take the null elements into account. The distinguishable undefined and null-elements were defined as follows:

$$I[\![\texttt{invalid}]\!]\tau \equiv \bot \quad \text{and} \quad I[\![\texttt{null}_{\texttt{Integer}}]\!]\tau \equiv \llcorner\bot\lrcorner \,.$$

An interpretation (consistent with [17]) is that $\texttt{null}_{\texttt{Integer}} + 3 = \texttt{invalid}$, and due to commutativity, we postulate $3 + \texttt{null}_{\texttt{Integer}} = \texttt{invalid}$, too. The necessary modification of the semantic interpretation looks as follows:

$$I[\![X + Y]\!]\,\tau \equiv \begin{cases} \llcorner\llcorner \ulcorner\ulcorner x \urcorner\urcorner + \ulcorner\ulcorner x \urcorner\urcorner \lrcorner\lrcorner & \text{if } x \neq \bot,\, y \neq \bot,\, \ulcorner x \urcorner \neq \bot \text{ and } \ulcorner y \urcorner \neq \bot \\ \bot & \text{otherwise}\,. \end{cases}$$

where $x = I[\![X]\!]\,\tau$ and $y = I[\![Y]\!]\,\tau$. The resulting principle here is that operations on the primitive types Boolean, Integer, Real, and String treat null as invalid (except $\_ \doteq \_$, $\_$`.oclIsInvalid()`, $\_$`.oclIsUndefined()`, casts between the different representations of `null`, and type-tests).

This principle is motivated by our intuition that invalid represents known errors, and null-arguments of operations for Boolean, Integer, Real, and String belong to this class. Thus, we must also modify the logical operators such that $\texttt{null}_{\texttt{Boolean}}$ `and false` $\triangleq$ `false` and $\texttt{null}_{\texttt{Boolean}}$ `and true` $\triangleq \bot$.

With respect to definedness reasoning, there is a price to pay. For most basic operations we have the rule:

$$\texttt{not}\;(X + Y)\;\texttt{.oclIsInvalid()} \triangleq (\texttt{not}\;X\;\texttt{.oclIsUndefined()}) \\ \texttt{and}\;(\texttt{not}\;Y\;\texttt{.oclIsUndefined()})$$

where the test x `.oclIsUndefined()` covers in fact two cases: x `.oclIsInvalid()` and $x \doteq$ `null` (i. e., x is `invalid` or `null`). As a consequence, for the inverse case $(X+Y)$ `.oclIsInvalid()`[3] there are four possible cases for the failure instead of two in the semantics described in [16]: each expression can be an erroneous `null`, or report an error. However, since all built-in OCL operations yield non-null elements (e. g., we have the rule `not` $(X + Y \doteq \texttt{null}_{\texttt{Integer}})$), a pre-computation can drastically reduce the number of cases occurring in expressions except for the base case of variables (e. g., parameters of operations and *self* in invariants). For these cases, it is desirable that implicit pre-conditions were generated as default, ruling out the `null` case. A convenient place for this are the multiplicities, which can be set to `1` (i. e., `1..1`) and will be interpreted as being non-null (see discussion in Section 5 for more details).

Besides, the case for collection types is analogously: besides an invalid collection, there is a $\texttt{null}_{\text{Set(T)}}$ collection as well as collections that contain null values (such as $\text{Set}\{\texttt{null}_{\text{T}}\}$) but never `invalid`.

[3] The same holds for $(X + Y)$ `.oclIsUndefined()`.

4.2 Null in Class Types

Of course, it is a viable option to rule out undefinedness in object-graphs *as such.* The essential source for such undefinedness are oid's which do not occur in the state, i. e., which represent "dangling references." Ruling out undefinedness as result of object accessors would correspond to a world where an accessor is either set explicitly to `null` or to a defined object; pragmatically, this corresponds to a discipline of constructors that initialize their arguments and the absence of an explicit deletion operation assuming a garbage collector as part of the underlying memory model (as, for instance, in Spec# [2]). Technically, this can be enforced by strengthening the state invariant inv_σ by adding clauses that state that in each object representation all oid's are either `nullid` or element of the domain of the state.

We deliberately decided against this option for the following reasons:

1. *methodologically* we do not like to constrain the semantics of OCL without clear reason; in particular, "dangling references" exist in C and C++ programs and it might be necessary to write contracts for them, and
2. *semantically*, the condition "no dangling references" can only be formulated with the complete knowledge over all classes and their layout in form of object representations. This restricts the OCL semantics to a closed world model.[4]

We can model `null`-elements as object-representations with `nullid` as their oid:

Definition 1 (Representation of `null`-Elements). *Let C_i be a class type with the attributes $A_1, \ldots, A_n$. Then we define its null object representation by:*

$$I[\![\mathtt{null}_{Ci}]\!]\tau \equiv {}_\llcorner(\mathtt{nullid}, \mathrm{arb}_t, a_1, \ldots, a_n)_\lrcorner$$

where the a_i are $\bot$ for primitive types and collection types, and `nullid` *for simple class types.* arb_t *is an arbitrary underspecified constant of the tag-type.*

Due to the outermost lifting, the null object representation is a defined value, and due to its special reference `nullid` and the state invariant, it is a typed value not "living" in the state. The $\mathtt{null}_\mathrm{T}$-elements are not equal, but isomorphic: Each type, has its own unique $\mathtt{null}_\mathrm{T}$-element; consequently, they can be mapped, i. e., casted, isomorphic to each other. In HOL-OCL, we can overload constants by parametrized polymorphism which allows us to drop the index in this environment.

The referential strict equality works as follows: we can now write *self* $\doteq$ `null` in OCL. Recall that $\_ \doteq \_$ is based on the projection OidOf from object-representations.

[4] In our presentation, the definition of `state` in Section 3 is also closed world. However, this limitation can be easily overcome by leaving "polymorphic holes" in our object representation universe, i. e., by extending the type sum in the state definition to $C_1 + \cdots + C_n + \alpha$. The details of the management of universe extensions are involved, but implemented in HOL-OCL and described in [5] in detail. However, these constructions exclude knowing the set of sub-oid's in advance.

4.3 Revised Accessors

Having introduced `null`-elements, the modification of the accessor functions is now straight-forward:

$$I[\![obj.\,\mathrm{a}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } I[\![obj]\!](\sigma,\sigma') = \bot \vee \mathrm{OidOf}\ulcorner I[\![obj]\!](\sigma,\sigma')\urcorner \notin \mathrm{dom}\ \sigma' \\ \mathtt{null}_{\mathrm{D}} & \text{if } \mathrm{get}_{\mathrm{C}}\ulcorner\sigma'(\mathrm{OidOf}\ulcorner I[\![obj]\!](\sigma,\sigma')\urcorner)\urcorner.\,\mathrm{a}^{(0)} = \mathtt{nullid} \\ \mathrm{get}_{\mathrm{D}}\ u & \text{if } \sigma'(\mathrm{get}_{\mathrm{C}}\ulcorner\sigma'(\mathrm{OidOf}\ulcorner I[\![obj]\!](\sigma,\sigma')\urcorner)\urcorner.\,\mathrm{a}^{(0)}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise.} \end{cases}$$

The definitions for type-cast and dynamic type test—which are not explicitly shown in this paper, see [5] for details—can be generalized accordingly. In the sequel, we will discuss the resulting properties of these modified accessors.

First of all, all functions of the induced signature are strict. This means that this holds for accessors, casts and tests, too:

$$\mathtt{invalid}.\,\mathrm{a} \triangleq \mathtt{invalid} \qquad \mathtt{invalid.oclAsType}(C) \triangleq \mathtt{invalid}$$
$$\mathtt{invalid.oclIsTypeOf}(C) \triangleq \mathtt{invalid}$$

Casts on `null` are always valid, since they have an individual dynamic type and can be casted to any other null-element due to their isomorphism.

$$\mathtt{null}_{\mathrm{A}}.\,\mathrm{a} \triangleq \mathtt{invalid} \qquad \mathtt{null}_{\mathrm{A}}\mathtt{.oclAsType}(B) \triangleq \mathtt{null}_{\mathrm{B}}$$
$$\mathtt{null}_{\mathrm{A}}\mathtt{.oclIsTypeOf}(A) \triangleq \mathtt{true}$$

for all attributes a and classes A, B, C where $C < B < A$. These rules are further exceptions from the standard's general rule that `null` may never passed as first ("*self*") argument.

4.4 Other Operations on States

Defining `_::allInstances()` is straight-forward; the only difference is the property T`::allInstances()->excludes(null)` which is a consequence of the fact that `null`'s are values and do not "live" in the state. In our semantics which admits states with "dangling references," it is possible to define a counterpart to `_.oclIsNew()` called `_.oclIsDeleted()` which asks if an objectid (represented by an object representation) is contained in the pre-state, but not the post-state.

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. This framing problem is well-known (one of the suggested solutions is [13]). We define:

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

where `S` is a set of object representations, encoding a set of oid's. The semantics of this operator is defined that any object whose oid is *not* represented in `S` the corresponding object representation will not change in the transition from pre-state to post-state:

$$I[\![X\texttt{->modifiedOnly()}]\!](\sigma,\sigma') \quad\equiv\quad \begin{cases}\bot & \text{if } X' = \bot \\ \llcorner\forall i \in M.\ \sigma\ i = \sigma'\ i\lrcorner & \text{otherwise}.\end{cases}$$

where $X' = I[\![X]\!](\sigma,\sigma')$ and $M = (\mathrm{dom}\ \sigma \cap \mathrm{dom}\ \sigma') - \{\mathrm{OidOf}\ x |\ x \in \ulcorner X' \urcorner\}$. Thus, if we require in a postcondition `Set{}->modifiedOnly()`, this means that an operation is a query in the sense of the OCL standard, i. e., the `isQuery` property is true. So, whenever we have $\tau \vDash$`X->modifiedOnly()` and $\tau \vDash$`X->excludes(s.a)`, we can infer that $\tau \vDash$`s.a = s.a@pre` (provided they are valid).

5 Attribute Values

The evaluation of an attribute for an object can yield a value or a collection of values. The type of the evaluation result depends on the multiplicity specified for the attribute. A multiplicity defines a lower bound as well as a possibly infinite upper bound on the cardinality of the attribute's values.

5.1 Single-Valued Attributes

If the upper bound specified by the attribute's multiplicity is one, then an evaluation of the attribute yields a single value. Thus, the evaluation result is not a collection. If the lower bound specified by the multiplicity is zero, the evaluation is not required to yield a non-null value. In this case an evaluation of the attribute can return `null` to indicate an absence of value.

To facilitate accessing attributes with multiplicity `0..1`, the OCL standard states that single values can be used as sets by calling collection operations on them. However, the implicit conversion of a value to a `Set` is not defined by the standard. We argue that the resulting set cannot be constructed the same way as when evaluating a `Set` literal. Otherwise, `null` would be mapped to the singleton set containing `null`, but the standard demands that the resulting set is empty in this case. The conversion should instead be defined as follows:

```
context OclAny::asSet():T
  post: if self ≐ null then result ≐ Set{}
        else result ≐ Set{self} endif
```

5.2 Collection-Valued Attributes

If the upper bound specified by the attribute's multiplicity is larger than one, then an evaluation of the attribute yields a collection of values. This raises the question whether `null` can belong to this collection. The OCL standard states that `null` can be owned by collections. However, if an attribute can evaluate to a collection containing `null`, it is not clear how multiplicity constraints should be interpreted for this attribute. The question arises whether the `null` element should be counted or not when determining the cardinality of the collection. Recall that `null` denotes the absence of value in the case of a cardinality upper bound of one, so we would assume that `null` is not counted. On the other hand, the operation `size` defined for collections in OCL does count `null`.

We propose to resolve this dilemma by regarding multiplicities as optional. This point of view complies with the UML standard, that does not require lower and upper bounds to be defined for multiplicities.[5] In case a multiplicity is specified for an attribute, i. e., a lower and an upper bound are provided, we require any collection the attribute evaluates to not contain `null`. This allows for a straightforward interpretation of the multiplicity constraint. If bounds are not provided for an attribute, we consider the attribute values to not be restricted in any way. Because in particular the cardinality of the attribute's values is not bounded, the result of an evaluation of the attribute is of collection type. As the range of values that the attribute can assume is not restricted, the attribute can evaluate to a collection containing `null`. The attribute can also evaluate to `invalid`. Allowing multiplicities to be optional in this way gives the modeler the freedom to define attributes that can assume the full ranges of values provided by their types. However, we do not permit the omission of multiplicities for association ends, since the values of association ends are not only restricted by multiplicities, but also by other constraints enforcing the semantics of associations. Hence, the values of association ends cannot be completely unrestricted.

5.3 The Precise Meaning of Multiplicity Constraints

We are now ready to define the meaning of multiplicity constraints by giving equivalent invariants written in OCL. Let `a` be an attribute of a class `C` with a multiplicity specifying a lower bound m and an upper bound n. Then we can define the multiplicity constraint on the values of attribute `a` to be equivalent to the following invariants written in OCL:

```
context C
  inv lowerBound: a->size() >= m
  inv upperBound: a->size() <= n
  inv notNull: not a->includes(null)
```

If the upper bound n is infinite, the second invariant is omitted. For the definition of these invariants we are making use of the conversion of single values to sets described in Section 5.1. If $n \leq 1$, the attribute `a` evaluates to a single value, which is then converted to a `Set` on which the `size` operation is called.

If a value of the attribute `a` includes a reference to a non-existent object, the attribute call evaluates to `invalid`. As a result, the entire expressions evaluate to `invalid`, and the invariants are not satisfied. Thus, references to non-existent

[5] We are however aware that a well-formedness rule of the UML standard does define a default bound of one in case a lower or upper bound is not specified.

```
package rbt
  context RBT
    inv wf: not left.oclIsInvalid() and not right.oclIsInvalid()
    inv redinv: color implies ((left ≐ null or not left.color)
                                and (right ≐ null or not right.color))
    inv ordinv: (left ≐ null or left.max() < key) and
                (right ≐ null or right.min() > key)
    inv balinv: black_depth(left) ≐ black_depth(right)

  context RBT::min():Integer
    post: if left≐null then key else left.max() endif

  context RBT::max():Integer
    post: if right≐null then key else right.max() endif

 -- Only count black nodes in left branch
 context RBT::black_depth(tree: RBT):Integer
    post: (tree ≐ null and result ≜ 0)
          or (tree.left.color and result ≐ black_depth(tree.left))
          or (not tree.left.color and result ≜ black_depth(tree.left) + 1)

  context RBT::isMember(tree: RBT, a:Integer):Boolean
    post: result ≜ tree <> null and (a ≐ tree.key or isMember(tree.left, a)
                                      or isMember(tree.right, a))

  context RBT :: subtrees():Set(RBT)
    post: result ≜ left->collect(subtrees())
                   ->union(right->collect(subtrees()))->asSet()

  context RBT::insert(k : Integer):
    post: subtrees()->modifiedOnly() and
          subtrees().key->asSet() ≐ subtrees@pre().key->asSet()->including(k)
endpackage
```

Listing 1.1. OCL specification of Red-black Trees.

objects are ruled out by these invariants. We believe that this result is appropriate, since we argue that the presence of such references in a system state is usually not intended and likely to be the result of an error. If the modeler wishes to allow references to non-existent objects, she can make use of the possibility described above to omit the multiplicity.

6 Example: Red-Black Trees

We give a small example to demonstrate how the semantics we presented for undefined values facilitates specification. In Figure 2 and Listing 1.1 describes a class for representing red-black trees. A red-black tree is a binary tree that satisfies an additional balancing invariant to ensure fast lookups. Each node of the tree is associated with a color (i. e., red or black) to allow for balancing.

Every instance of the tree class represents a red-black tree. The empty tree is represented by `null`. A tree object is connected to its left and right subtrees via associations. The class also has the attribute `key` for storing the data and the attribute `color` for indicating the node color.

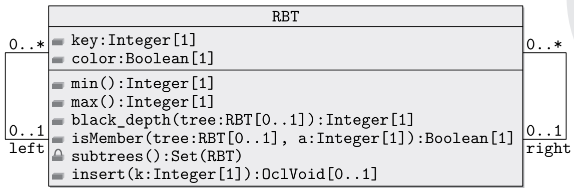

Figure 2. A class representing red-black trees.

The availability of the `null` value for representing the empty tree clearly simplifies the specification. Constructing a dummy tree to denote the empty tree would certainly be a burden, and it would also be necessary to assign dummy data to the key field. The only other alternative would be to represent the empty tree by the other undefined value `invalid`. However, it is easy to see that this choice would also obscure the specification substantially. Recall that every operation call with an `invalid` argument evaluates to `invalid`, so the tree operations could not be called for the empty tree. Instead, the case of an empty tree would always have to be considered additionally. In the postcondition of the operation `isMember`, for example, the two recursive calls to `isMember` would require two tests for the empty tree, which would increase the size of the postcondition considerably.

For the postcondition of `insert` we make use of `_->modifiedOnly()` that we introduced in Section 4. We use this construct to state that the only objects the operation may modify are the subtrees of the tree that the operation is called for. Without this constraint it would not be guaranteed that the operation does not modify other unrelated trees or even other objects of a completely different type. Thus, `_->modifiedOnly()` allows us to express properties that are essential for the completeness of the specification.

Another advantage of our semantics is that references to non-existent objects can easily be ruled out a priori by the invariant `wf`.[6] Hence, it is guaranteed that every non-null tree object encountered during a search is a valid subtree and not a dangling reference. This property is essential for the specification correctness.

7 Discussion

We have presented a formal semantics for OCL 2.1 as an increment to the machine-checked HOL-OCL semantics presented in textbook format. The achievement is a proposal how to handle null-elements in the specification language which result from the current attempt to align the UML infrastructure [18] with the OCL standard; an attempt that has great impact on both the semantics of UML and, to an even larger extent, OCL. Inconsistencies on the current standardization documents as result of an ad-hoc integration have been identified as a major obstacle in OCL tool development. We discussed the consequences of the integrated semantics by presenting the *derived rules*, their *implications for multiplicities*, and their *pragmatics* in a non-trivial example, which shows how `null` elements can help to write concise, natural, design-level contracts for

[6] In fact, the invariant `wf` is redundant since it is implied by the multiplicity constraints (see Section 5). The multiplicity constraints of the attributes `key` and `color` ensure that these attributes are neither `null` nor `invalid`.

object-oriented code in a programming like style. Adding a basic mechanism to express framing conditions gives the resulting language a similar expressive power as, for example, JML or Spec#.

7.1 Related Work

While null elements are a common concept, e. g., in programming languages or database design, there are, to our knowledge, no proposals at all for a formal semantics of null elements in the context of OCL. Albeit, there are object-oriented specification languages that support null elements, namely JML [15] or Spec# [2]. Notably, both languages limit null elements to class types and provide a type system supporting non-null types. In the case of JML, the non-null types are even chosen as the default types [7]. Supporting non-null types simplifies the analysis of specifications drastically, as many cases resulting in potential invalid states (e. g., de-referencing a null) are already ruled out by the type system.

Our concept for modeling frame properties is essentially identical (but simpler) to [13], 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; an idea motivated by the presence of a garbage collector).

7.2 Future Work

Of course, there are numerous other concepts in the current OCL definition that deserve formal analysis; for example, the precise notion of signals, method overriding, overload-resolution, recursive definitions, and the precise form of interaction between class models, state machines and sequence charts.

However, from the more narrower perspective of this work on integrating `null` elements, adding non-null types and a non-null type inference to OCL (similar to [9, 10]) seems to be the most rewarding target.

References

[1] P. B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, 2002.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, eds., *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, *LNCS*, vol. 3362, pp. 49–69. Springer-Verlag, 2005.

[3] A. D. Brucker, J. Doser, and B. Wolff. An MDA framework supporting OCL. *Electronic Communications of the EASST*, 5, 2006.

[4] A. D. Brucker and B. Wolff. The HOL-OCL book. Tech. Rep. 525, ETH Zurich, 2006.

[5] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in HOL. *Journal of Automated Reasoning*, 41, 2008.

[6] A. D. Brucker and B. Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 2009. Accepted for publication.

[7] P. Chalin and F. Rioux. Non-null references by default in the java modeling language. In *Proceedings of the conference on Specification and verification of component-based systems*, p. 9. ACM Press, 2005.

[8] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[9] T. Ekman and G. Hedin. Pluggable checking and inferencing of nonnull types for Java. *Journal of Object Technology*, 6(9):455–475, 2007.

[10] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In R. Crocker and G.L. Steele Jr., eds., *OOPSLA*, pp. 302–312. ACM, 2003.

[11] M. Gogolla, M. Kuhlmann, and F. Büttner. A benchmark for OCL engine accuracy, determinateness, and efficiency. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, eds., *MoDELS*, *LNCS*, vol. 5301, pp. 446–459. Springer-Verlag, 2008.

[12] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[13] P. Kosiuczenko. Specification of invariability in OCL. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, eds., *Model Driven Engineering Languages and Systems (MoDELS)*, *LNCS*, vol. 4199, pp. 676–691. Springer-Verlag, 2006.

[14] 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.

[15] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual (revision 1.2). 2007.

[16] UML 2.0 OCL specification. 2003. Available as ptc/03-10-14.

[17] UML 2.0 OCL specification. 2006. Available as formal/06-05-01.

[18] UML 2.2 infrastructure specification. 2009. Available as formal/2009-02-04.

[19] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. Ph.D. thesis, Universität Bremen, Logos Verlag, BISS Monographs, No. 14, 2002.