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.

Keywords: HOL-OCL, UML, OCL, null reference, formal semantics

1 Introduction

The Object Constraint Language (OCL) is used for specifying 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` in [17]) 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 can be handled

* This work was partially supported by the Digiteo Foundation.

S. Ghosh (Ed.): MODELS 2009 Workshops, LNCS 6002,, pp. 261–275, 2010.

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. This second exception element, called `null`, is intended to represent the *absence of value*. 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. 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 need to check for `null`. 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` for representing the absence of a value. 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 state clearly when object attributes can evaluate to `null` and how this depends on its multiplicity. The standard does also not clarify whether objects that do not exist ("dangling references") are treated the same way as `null` or not.

Our contribution is a proposal for a formal semantics that overcomes these problems in the current version of the OCL standard. From this semantics, we derive formally and informally numerous rules, which could be included in the mandatory part of a standardization document, while our semantics could serve as foundation of a future "Annex A." We proceed as follows: In Section 3, we provide a summary of the essentials of the HOL-OCL semantics as it could be found in semantics textbooks (rather than a technical presentation motivated by machine readable documents). Nevertheless, our semantics 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 on 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.

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*. 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 $[\alpha \Rightarrow \text{bool}] \Rightarrow$ bool. The quantifiers may range over types of higher order, i. e., functions or sets. The definition of the element-hood $\_ \in \_$, the set comprehension $\{\_.\_\}$, as well as $\_ \cup \_$ and $\_ \cap \_$ are standard.

The Isabelle/HOL library [15] contains formal definitions and theorems for practically all mathematical concepts used in computer science, including typed set theory, well-founded recursion theory, number theory and theories for datastructures like Cartesian products $\alpha \times \beta$ and disjoint type sums $\alpha + \beta$. The library also includes the type constructor $\tau_\perp := \perp \mid \llcorner\_\lrcorner : \alpha$ that assigns to each type τ a type $\tau_\perp$ *disjointly extended* by the exceptional element $\perp$. The function $\ulcorner\_\urcorner : \alpha_\perp \Rightarrow \alpha$ is the inverse of $\llcorner\_\lrcorner$ (unspecified for $\perp$). Partial functions $\alpha \rightharpoonup \beta$ are defined as functions $\alpha \Rightarrow \beta_\perp$ supporting the usual concepts of domain (dom $\_$) and range (ran $\_$). 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 and on Isabelle/HOL. HOL-OCL also reuses and extends the existing Isabelle front-end called Proof General as well as the Isabelle documentation generator.

3 An Overview over OCL Semantics

In this section, we will briefly introduce 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 that the latter is a machine-checked, "strong" formal semantics which is itself based on a typed meta-language (i. e., HOL) instead of an untyped one (i. e., naïve set theory), and various technical simplifications: instead of three different semantic interpretation functions $I(x)$, $I[\![e]\!]\tau$, $I_{\text{ATT}}[\![e]\!]\tau$, 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 (for details, see [6]).

3.1 Validity and Evaluations

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

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

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 $\bot$ denoting undefinedness, we define formally:

$$(\sigma, \sigma') \models P \equiv \big(P(\sigma, \sigma') = \lfloor\!\lceil \mathrm{true} \rceil\!\rfloor\big)\,.$$

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 $\tau_\bot$. All types of expressions are of a form captured by

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

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 $\_.\mathrm{a}$ by their counterparts $\_.\mathrm{a}\,\texttt{@pre}$. For example, $(\mathit{self}.\mathrm{a} > 5)_{\mathrm{pre}}$ is just $(\mathit{self}.\mathrm{a}\,\texttt{@pre} > 5)$.

3.2 Strict Operations

An operation is called strict if it returns $\bot$ if one of its arguments is $\bot$. Most OCL operations are strict, e. g., the Boolean negation is formally presented as:

$$I[\![\texttt{not } X]\!]\tau \equiv \begin{cases} \lfloor\!\lceil \neg \lceil I[\![X]\!]\tau \rceil \rceil\!\rfloor & \text{if } I[\![X]\!]\tau \neq \bot, \\ \bot & \text{otherwise}\,. \end{cases}$$

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 constructs, i. e., `not _` are HOL functions (in this case of HOL type $V(\mathrm{bool}) \Rightarrow V(\mathrm{bool})$) that can be viewed as *transformers on evaluations*.

The binary case of the integer addition is analogous:

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

Here, the operator $\_ + \_$ on the right refers to the integer HOL operation with type $[\text{int}, \text{int}] \Rightarrow \text{int}$. The type of the corresponding strict HOL-OCL operator $\_ + \_$ is $[V(\text{int}), V(\text{int})] \Rightarrow V(\text{int})$. A slight variation of this definition scheme is used for the operators on collection types such as HOL-OCL sets or sequences:

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

Here, S ("smash") is a function that maps a lifted set $\lfloor X \rfloor$ 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 `E` of HOL-OCL type `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 $X\,\texttt{.oclIsInvalid()} \equiv (X \triangleq \texttt{invalid})$. 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: $X\ \texttt{or}\ \ Y \equiv \texttt{not}\,(\texttt{not}\ X\ \texttt{and not}\ Y)$ and $X\ \texttt{implies}\ Y \equiv \texttt{not}\ X\ \texttt{or}\ \ Y$.

3.4 Object-oriented Data Structures

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 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 (see [5] for the complete model).

First, object identifiers are captured by 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 administration information and the information for all attributes of an object; here, the primitive types 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). Third, the class type C will be the type of such an object representation: $C := (\mathrm{oid} \times \mathrm{C_t} \times A_1 \times \cdots \times A_k)$ where a unique tag-type $\mathrm{C_t}$ (ensuring type-safety) is created for each class type, where the types $A_1, \ldots, A_k$ are the attribute types (including inherited attributes) with class types substituted by oid. The function OidOf projects the first component, the oid, out of an object representation. Fourth, 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: $\forall \mathit{oid} \in \mathrm{dom}\, f.\ \mathit{oid} = \mathrm{OidOf}\,\lceil f(\mathit{oid})\rceil$. 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 primitive type, not a collection):

$$I[\![\mathit{self}.\,\mathrm{a}]\!](\sigma, \sigma') \equiv \begin{cases} \bot & \text{if } O = \bot \vee \mathrm{OidOf}\,\lceil O\rceil \notin \mathrm{dom}\ \sigma' \\ \mathrm{get_D}\ u & \text{if } \sigma'(\mathrm{get_C}\lceil\sigma'(\mathrm{OidOf}\,\lceil O\rceil)\rceil.\,\mathrm{a}^{(0)}) = \lfloor u \rfloor, \\ \bot & \text{otherwise.} \end{cases}$$

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

where $O = I[\![self]\!](\sigma,\sigma')$. 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). For simple class types, we have to evaluate expression *self*, get an object representation (or undefined), project the attribute, de-reference it in the pre or post state 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 primitive 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. Thus, a dangling reference, i. e., one that is *not* in dom σ, 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 (underlying HOL-OCL and essentially formalizing [16, Annex A]). In later versions of the standard [17] the formal semantics appendix reappears although being incompatible with the normative parts of the standard. Not all rules shown here are formally proven; technically, these are informal proofs "with a glance" on the formal proofs shown in the previous section.

4.1 Revised Operations on Primitive Types

In UML, and since [17] in OCL, all primitive types comprise the `null`-element, modeling the possibility to be non-existent. 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 results in lifting any primitive type twice:

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

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

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

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:

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

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 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 $\texttt{not}\,(X + Y \doteq \texttt{null}_{\texttt{Integer}})$), 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 analogous: in addition to the invalid collection, there is a $\texttt{null}_{\mathrm{Set(T)}}$ collection as well as collections that contain null values (such as $\mathrm{Set}\{\texttt{null}_{\mathrm{T}}\}$) but never `invalid`.

4.2 Null in Class Types

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 always set explicitly to `null` or to a defined object; in a programming language without explicit deletion and where constructors always initialize their arguments (e. g., Spec# [2]), this may suffice. Semantically, this can be modeled 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.

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

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 of 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:*

$$I[\![\texttt{null}_{Ci}]\!]\tau \equiv \lfloor(\texttt{nullid}, \text{arb}_t, a_1, \ldots, a_n)\rfloor$$

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 $\texttt{null}_\text{T}$-elements are not equal, but isomorphic: Each type, has its own unique $\texttt{null}_\text{T}$-element; 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 allows us to write *self* $\doteq$ `null` in OCL. Recall that $\_ \doteq \_$ is based on the projection OidOf from object-representations.

4.3 Revised Accessors

The modification of the accessor functions is now straight-forward:

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

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.

[4] In our presentation, the definition of `state` in Section 3 assumes a closed world. 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 (see [5] for details). However, these constructions exclude knowing the set of sub-oid's in advance.

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

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

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.

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

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 be 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 object id (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 such that for any object whose oid is *not* represented in `S` and that is defined in pre and post state, the corresponding object representation will not change in the state transition:

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

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()` and exclude via `_.oclIsNew()` and `_.oclIsDeleted()` the existence of new or deleted objects, the 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` (if they are valid).

5 Attribute Values

Depending on the specified multiplicity, the evaluation of an attribute can yield a value or a collection of values. 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. This 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 to not contain `null`. This allows for a straightforward interpretation of the multiplicity constraint. If bounds are not

[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.

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 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. Figure 1 and Listing 1.1 describe 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 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 data is stored in the attribute `key` and the node color in the attribute `color`.

```
  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

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)
```

Listing 1.1. OCL specification of red-black trees.

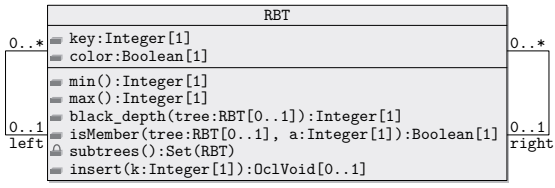

Fig. 1. A class representing red-black trees.

The availability of the `null` value for representing the empty tree clearly simplifies the specification. One might object here that one could alternatively introduce a subclass `EmptyTree` inheriting from `RBT`; however, this implies that there are `EmptyTree`-objects which have to be allocated and deallocated resulting in a different memory behavior. In contrast, representing empty trees by `null` allows for specifying tree operations as they are commonly realized in Java or C++, i. e. our extensions pave the way for *design level contracts* in OCL.

The only remaining alternative would be to represent the empty tree by the other undefined value `invalid`. However, it is easy to see that this choice would 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.

The postcondition of `insert` uses `_->modifiedOnly()` (see Section 4) for stating 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 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 [14] 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).

[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`.

7.2 Future Work

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 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. Andrews, P.B.: Introduction to Mathematical Logic and Type Theory: To Truth through Proof, 2nd edn. Kluwer Academic Publishers, Dordrecht (2002)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: G. Barthe, L. Burdy, M. Huisman, J.L. Lanet, T. Muntean (eds.) Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS), *LNCS*, vol. 3362, pp. 49–69. Springer-Verlag (2005).
3. Brucker, A.D., Doser, J., Wolff, B.: An MDA framework supporting OCL. Electronic Communications of the EASST **5** (2006).
4. Brucker, A.D., Wolff, B.: The HOL-OCL book. Tech. Rep. 525, ETH Zurich (2006).
5. Brucker, A.D., Wolff, B.: An extensible encoding of object-oriented data models in HOL. Journal of Automated Reasoning **41**, 219–249 (2008).
6. Brucker, A.D., Wolff, B.: Semantics, calculi, and analysis for object-oriented specifications. Acta Informatica **46**(4), 255–284 (2009).
7. Chalin, P., Rioux, F.: Non-null references by default in the Java modeling language. In: SAVCBS '05: Proceedings of the 2005 conference on Specification and verification of component-based systems, p. 9. ACM Press (2005).
8. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5**(2), 56–68 (1940)
9. Ekman, T., Hedin, G.: Pluggable checking and inferencing of nonnull types for Java. Journal of Object Technology **6**(9), 455–475 (2007)
10. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: OOPSLA, pp. 302–312. ACM Press (2003).
11. Gogolla, M., Kuhlmann, M., Büttner, F.: A benchmark for OCL engine accuracy, determinateness, and efficiency. In: K. Czarnecki, I. Ober, J.M. Bruel, A. Uhl, M. Völter (eds.) MoDELS, *LNCS*, vol. 5301, pp. 446–459. Springer-Verlag (2008).
12. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press (1993)
13. Kosiuczenko, P.: Specification of invariability in OCL. In: O. Nierstrasz, J. Whittle, D. Harel, G. Reggio (eds.) Model Driven Engineering Languages and Systems (MoDELS), *LNCS*, vol. 4199, pp. 676–691. Springer-Verlag (2006).
14. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D.R., Müller, P., Kiniry, J., Chalin, P.: JML reference manual (revision 1.2) (2007).
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle's logic: HOL (2009)
16. UML 2.0 OCL specification (2003). Available as OMG document ptc/03-10-14
17. UML 2.0 OCL specification (2006). Available as OMG document formal/06-05-01
18. UML 2.2 infrastructure (2009). Available as OMG document formal/2009-02-04
19. Richters, M.: A precise approach to validating UML models and OCL constraints. Ph.D. thesis, Universität Bremen, Logos Verlag, BISS Monographs, No. 14 (2002)

A Alternative Specifications of Red-Black-Trees[7]

In this appendix we give alternatives to our red-black tree specification in Section 6. Our objective is to show the advantages and disadvantages of different ways of representing the empty tree.

A.1 Representing the Empty Tree by `invalid`

The specification in Listing 1.2 represents the empty tree by `invalid`. This shows how such a specification could be written without `null` if `null` were not available in OCL[8]. The disadvantage is that every operation call with an `invalid` argument evaluates to `invalid`, so the tree operations can not be called for the empty tree. Instead, the case of an empty tree always has to be considered additionally. In the specification of the operation `black_depth`, we use an if expression to check for empty trees. This clearly is a nuisance. It would be much nicer if we could call the operation `black_depth` directly on potentially empty trees instead of introducing the intermediate variable `left_depth`. The specification of `isMember` is even more problematic since it contains two recursive operation calls, each requiring a test for empty trees.

```
context RBT::black_depth():Integer
  post: let left_depth: Integer =
           if left.oclIsInvalid() then 0 else left.black_depth() endif
        in left.color and result ≙ left_depth
           or not left.color and result ≙ left_depth + 1

context RBT::isMember(a:Integer):Boolean
  post: result ≙ (a ≐ tree.key
        or not left.oclIsInvalid() and left.isMember(a)
        or not right.oclIsInvalid() and right.isMember(a))
```

Listing 1.2. Representing the empty tree by `invalid`.

A.2 Representing the Empty Tree by a Specialized Class

The specification in Figure 2 and Listing 1.3 is an attempt to describe red-black trees more elegantly by avoiding these problems. A new class `EmptyTree` is introduced in order to model empty trees. Empty trees are represented by an object of this class. It is now possible to keep the postconditions of the operations `black_depth` and `isMember` more succinct, at the cost of spreading the specification across two classes. A disadvantage of this specification is that an implementation derived from it is likely to be less efficient than the original

[7] This appendix extends the version published by Springer-Verlag.

[8] This specification is not correct with respect to the semantics defined in this paper, since in this semantics attributes constrained by multiplicities never evaluate to `invalid`. Our objective is to show what a specification would look like if a different semantics for OCL without `null` is chosen.

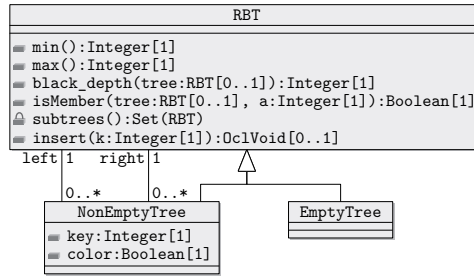

Fig. 2. Representing the empty tree by a specialized class.

specification in Section 6. An additional object needs to be allocated for the empty tree, and in general virtual function calls to `black_depth` and `isMember` incur an overhead compared to direct calls.

```
context NonEmptyTree::black_depth():Integer
  post: left.color and result ≜ left.black_depth()
        or not left.color and result ≜ left.black_depth() + 1

context NonEmptyTree::isMember(a:Integer):Boolean
  post: result ≜ (a ≐ tree.key or left.isMember(a) or right.isMember(a))

context EmptyTree::black_depth():Integer
  post: result ≜ 0

context EmptyTree::isMember(a:Integer):Boolean
  post: result ≜ false
```

Listing 1.3. Representing the empty tree by a specialized class.