# Null Considered Harmful (for Transformation Verification)

K. Lano

Dept. of Informatics, King's College London, UK

**Abstract.** The use of explicit null and invalid values in OCL can lead to complex and hard-to-verify specifications. In addition, these values complicate the logic of OCL and of transformation languages that use OCL, making it difficult to provide effective verification support for these languages. We define an alternative technique for using OCL with UML and model transformations which avoids the use of null and undefined values, and we present verification techniques for a transformation language, UML-RSDS, based on this approach.

## 1 Introduction

OCL is the official textual specification language used with the UML, and it is also widely used as a constraint language within model transformation languages, such as ATL, QVT, ETL, Kermeta and others to define transformation rules, although each of these languages tend to adopt variations with the OCL standard. For example, Kermeta uses the keyword *void* for the null value, and ETL uses *isDefined*() instead of *oclIsUndefined*() in its variant OCL, EOL.

The OCL standard [13] defines a type *OclVoid* which has a single instance, *null*. *OclVoid* is a subtype of all other OCL types (except *OclInvalid*, which contains the *invalid* element), so that each of these types, including Boolean and Integer, also have *null* elements. *null* represents the absence of a valid value, in contrast to *invalid*, which represents an invalid evaluation. Confusingly, *null* is sometimes used as the value of expressions which would normally be considered invalid (eg., `1/0 = null` on page 17 of [3]).

The principal reason why *null* arises in UML and OCL is the need to represent the value of optional association ends (e.g., Figure 1): if a person has no spouse, this is expressed by $spouse = null$. An attempt to evaluate a feature of a *null* object results in *invalid*, however *null* is treated as being $Bag\{\}$ when a collection operation is applied to it. Thus $spouse.age$ is undefined if *spouse* is *null*, but $spouse \rightarrow collect(age)$ is $Bag\{\}$, contradicting the usual expectation that these expressions should have the same value.

Null values can also be returned by operations, and this leads to a problem which OCL shares with object-oriented programming languages: that programmers may return *null* from an operation to indicate that it has failed, instead of using exceptions or other error-handing facilities. For example an operation to

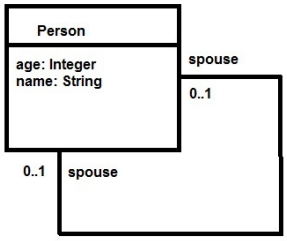


**Fig. 1.** Example of optional association ends

marry two *Person* objects could return *null* if the operation cannot be successfully completed, but this does not inform the caller why the operation failed: it can fail for several different reasons.

The explicit use of *null* therefore complicates system specification and verification by introducing undefinedness into expressions and by encouraging the use of hard-to-verify specification styles. It also has a direct impact on the use of verification tools by complicating the logic used for reasoning about systems and models. The truth tables for each logical operator *not*, *and*, *or*, *xor* and *implies* of OCL must incorporate cases for *null*, for example:

```
null and null   =  null
false and null  =  false
true and null   =  null
null and false  =  false
null and true   =  null
```

*null* acts as an unspecified or 'unknown' element in these tables, resulting in a 3-valued logic [3]. Additionally, the value *invalid* combined with any logical operater results in *invalid*, since a logical expression is invalid if any argument of its logical operators is invalid. To use verification tools which work in classical 2-valued logic, such as B [4] or Z3 [17], an analyst must therefore model this 3 or 4-valued OCL logic within the tool and formulate a specialised inference system in order to reason about OCL specifications that use *null*. This may lead to inefficiency and reduce the effectiveness of proof techniques, which are usually optimised for the inbuilt logic of the tool. In addition, the more complex 3 or 4-valued logic can be difficult for specifiers and analysts to understand and reason with.

Even within the UML standards there remain unclear issues with the semantics of *null* and *invalid*, for example, whether an *allInstances*() collection can contain an undefined element, or whether $null.oclIsKindOf(A)$ should return true/false or *invalid* (http://www.omg.org/issues/ocl2-rtf.open.html). Languages based on OCL often adopt variant semantics to the standard. For exam-

ple, whilst OCL permits *null* elements in collections, in QVT-O assignments *null* elements in collections are ignored (Section 8.2.2.11 of [14]).

This lack of clear semantics for *null* and *invalid* means that tools which do attempt to handle full OCL necessarily make specific assumptions about the semantics, which may not match the specifier's intentions. For example, in order to formally model *null* and *invalid* in HOL, [2] proposes that these values may occur in attribute values only if no multiplicity bounds are defined for the attribute.

We consider that this lack of a clear semantics, and the variability of definitions in different MT languages is an indication that the use of explicit *null* and *undefined* elements is unsuited to situations where formal verification of model transformations is required.

In this paper we propose an alternative approach for the use of OCL and related constraint languages in model transformation specifications, in accordance with the following principles:

> (1) Transformation specifications should use expressions which are ensured by their context to be well-defined (not *null* or *invalid*) and determinate in value. *null* or *invalid* values should not be used.
> (2) Expressions which are used as conditional tests or pure values should be non-side-effecting.

These principles greatly facilitate analysis and verification of model transformations, by ensuring that expressions can be given consistent meanings in (i) the transformation language; (ii) in formal languages used to analyse the transformation, and (iii) in programming languages used to implement the transformation.

In Section 2 we describe our proposed alternative to standard OCL, in Section 3 we describe how it can be used for transformation specification, and in Section 4 describe transformation verification based on alternative OCL.

## 2 Alternative OCL

In order to generally resolve the difficulties caused by explicit *null* and *invalid* values, we have developed and applied an alternative OCL-like formalism, which retains most of the notation and concepts of OCL, but which has no *null* or *invalid* elements, and is based on classical 2-valued logic and set theory. Transformation specifiers should ensure that invalid expression evaluations cannot occur in their transformations, using the definedness conditions of expressions (Table 1). In cases where operators can produce invalid evaluations, additional operators are provided to test for validity before the evaluation is attempted. Eg., for *toInteger*() on String we provide an operation *isInteger*() to check that a string represents a valid integer before attempting to convert it [11].

Side-effects are only permitted for expressions which are calls $obj.op(e)$ of update (non-query) operations $op$. Expressions should not have side-effects when used as conditions or pure values. Together with the restriction that undefined values cannot occur, this leads to the same logic for the logical connectives both

in OCL and in the formal and programming languages which we treat (B AMN, Java, C# and C++): in programming languages 'short-circuit' evaluation is used to determine that $P$ *and* $Q$ is false if $P$ is false, without the need to evaluate $Q$. This is semantically equivalent to an 'evaluate all arguments completely' semantics if undefined values and side effects are prohibited.

We use some minor extensions of OCL. For entity types $E$ with one attribute *id* designated as an identity attribute (primary key), the abbreviation $E[v]$ is introduced to represent the standard OCL expression $E.allInstances()\rightarrow any(id = v)$, if $v$ is single-valued, and to represent $E.allInstances()\rightarrow select(v\rightarrow includes(id))$ if $v$ is collection-valued. For an instance-scope attribute *att* of $E$, $E.att$ denotes $E.allInstances()\rightarrow collect(att)$.

Examples of the clauses for the definedness function $def : Exp(L) \rightarrow Exp(L)$ are given in Table 1. Expression determinacy $det : Exp(L) \rightarrow Exp(L)$ can be similarly defined.

| *Constraint expression e* | *Definedness condition def(e)* |
|---|---|
| $a/b$ | $b \neq 0$ *and* $def(a)$ *and* $def(b)$ |
| $e.f$<br>Data feature application | $def(e)$ *and* $E.allInstances()\rightarrow includes(e)$<br>where $E$ is the declared classifier of $e$ |
| Operation call $e.op(p)$ | $def(e)$ *and* $E.allInstances()\rightarrow includes(e)$ *and*<br>$def(p)$ *and* $def(e.Post_{op}(p))$ where $E$ is the declared classifier of $e$, $Post_{op}$ the postcondition of *op* |
| $s\rightarrow at(ind)$<br>sequence, string $s$ | $ind > 0$ *and* $ind \leq s\rightarrow size()$ *and*<br>$def(s)$ *and* $def(ind)$ |
| $E[v]$<br>entity type $E$ with identity attribute *id*, $v$ single-valued | $E.id\rightarrow includes(v)$ *and* $def(v)$ |
| $A$ *and* $B$<br>$A$ *implies* $B$ | $def(A)$ *and* $def(B)$<br>$def(A)$ *and* ($A$ *implies* $def(B)$) |
| $str.toInteger()$<br>$str.toReal()$<br>$obj.oclAsType(E)$ | $def(str)$ *and* $str.isInteger()$<br>$def(str)$ *and* $str.isReal()$<br>$def(obj)$ *and* $obj.oclIsKindOf(E)$ |

**Table 1.** Definedness conditions for expressions

Inductive proof of definedness may be needed for recursive operation calls.

The main difference between our constraint language and standard OCL is that optional association ends are treated as being *collections* of size 0 or 1, rather than being *null* or a single defined object. This interpretation is consistent with the use of multiplicity bounds for other many-valued association ends, and is also already adopted in OCL in the case that collection operators are applied to the optional association end (because of the implicit casting of *null* to $Bag\{\}$ in such cases). The test $spouse\rightarrow isEmpty()$ can then be used in place of $spouse.oclIsUndefined()$, and $spouse\rightarrow any(true)$ returns the *Person* object if *spouse* is non-empty.

This approach has several advantages:

- Classical logic can be used in reasoning, there is no need for *null* and *invalid*.
- It is simple to evolve a model using a 0..1 multiplicity association end to one with a 0..$n$ or * multiplicity, and vice-versa, since the association end remains a collection.
- Collections cannot contain *null* elements, simplifying their processing, both in OCL and in programs synthesised from the OCL.

In our alternative OCL semantics there is no *null* element, however there are 'unallocated object references'. There is a fixed denumerable type $Object\_OBJ$ of possible object references, a finite set $objects \subseteq Object\_OBJ$ of object references of existing objects, and for each class or interface $E$ in a UML class diagram, a finite set, $E.allInstances() \subseteq objects$, of the references that refer to actually existing $E$ objects. Creation of an $E$ instance selects an element $ex$ from $Object\_OBJ - objects$, adds it to $E.allInstances()$, $objects$ and to all superclass instance sets of $E$, and allocates an object for $ex$, including values for all features of $E$ (including inherited features). On deletion $ex \rightarrow isDeleted()$, the reverse process applies, and $ex$ is returned to the set of available object references. It is removed from any association end or other collection in the system. It is a semantic error to refer again to $ex$ after it has been deleted.

Applying our approach to the red-black tree specification of [2] yields a more concise specification, eg., the *redinv* can be expressed as:

$$color \;\; implies \;\; left \rightarrow forAll(not\ color) \;\; and \;\; right \rightarrow forAll(not\ color)$$

in comparison to

$$color \;\; implies \;\; ((left = null \;\; or \;\; not \;\; left.color) \;\; and \;\; (right = null \;\; or \;\; not \;\; right.color))$$

in [2].

OCL was originally intended as a language for logical expression evaluation, to express declarative constraints of UML models. However, transformation languages such as QVT and Kermeta now use OCL to define executable effects, to update models. The imperative use of OCL as a programming language introduces semantic problems, for example, how a *forAll* or other iteration should behave if (logically) its evaluation can be terminated before all elements are processed, whilst (imperatively) some intended side-effect behaviour could still arise if additional elements are considered. Failing to evaluate all of the quantified collection would also violate the requirement that the expression should evaluate to *invalid* if any of its individual element evaluations do.

We rule out such cases by clearly distinguishing the logical and imperative use of OCL. A function *stat* gives an imperative denotation of certain expressions as statements (activities). This defines precisely when an expression $e$ is interpreted imperatively, i.e., as an activity $stat(e)$ [10]. We exclude side-effects when an expression is used logically. Thus the logical evaluation of iterators can be made more efficient. In an imperative interpretation $stat(e)$ of expression $e$, however,

the computation only terminates when $e$ is established (which may require fixed-point iteration, etc.).

For example, logical evaluation of $s{\rightarrow}forAll(x \mid x.att = v)$ can return *false* as soon as some $x \in s$ with $x.att \neq v$ is found, and no updates are performed. But $stat(s{\rightarrow}forAll(x \mid x.att = v))$ is $for\ x : s\ do\ x.att := v$ and always iterates completely through all elements of $s$, assigning $v$ to $x.att$ for each $x \in s$. The logical and imperative interpretations are related naturally by the property

$$def(e) \Rightarrow [stat(e)]e$$

where [ ] is the weakest-precondition operator on activities.

## 3 Transformation specification

The alternative OCL described in the preceding section can be used to define model transformations at a declarative level. In UML-RSDS [11], such OCL-style constraints define pre and postconditions of transformations, represented as UML use cases. Each use case (transformation) $\tau$ has assumptions $Asm$ on the starting state of the models that it operates on, and a sequence $Cons$ of postconditions, which also define the imperative behaviour of the use case as a sequential composition of the activities $stat(Ci)$ of the postconditions $Ci$.

The typical form of a postcondition $Cn$ in $Cons$ is an implication constraint

$$Ante\ implies\ Succ$$

on a context entity type $ST$ belonging to the source language $S$ of $\tau$. This represents a declarative assertion that, at termination of the transformation, for each instance *self* of $ST$ which satisfies $Ante$, $Succ$ also holds. The definedness condition $def(Cn)$ from Table 1 is

$$def(Ante)\ and\ (Ante\ implies\ def(Succ))$$

so that the context $ST$ of $Cn$ must ensure that $Ante$ is defined, and that $Ante$ ensures definedness of $Succ$. The succedent should normally have an imperative denotation $stat(Succ)$. The definedness of each constraint in $Cons$ is a proof obligation which the transformation specifier should establish. $Asm$ and preceding postconditions can be used as assumptions for the definedness obligations. These obligations can be submitted to a proof tool. Likewise, the determinacy condition $det(Cn)$ of each postcondition should normally be established for the specification, and specifiers should ensure that no update (non-query) operation is invoked in any $Ante$ condition or other conditional test/value expression in any $Cn$. Whilst these obligations are additional proof burdens for the developer, they can help to identify errors in the specification, and provide assurance of correctness.

For separate-models transformations, $ST$ and other source model data are not modified by $\tau$, instead instances of target language entity types are created

and their features defined. A typical postcondition constraint for this kind of transformation has the form

$$Ante \ implies \ TT \rightarrow exists(t \mid Succ1)$$

which specifies that corresponding target model instances $t$ of target entity type $TT$ should exist for each source model instance *self* of $ST$ that satisfies $Ante$. $Succ1$ can define the feature values of $t$ based upon those of *self* (or on any other source or target model data).

Executable code in Java, C# and C++ can be generated for transformations (or for general software systems) from UML-RSDS specifications. A postcondition constraint $Cn$ is implemented by $stat(Cn)$, and this activity serves as an intermediate representation from which program code in specific programming languages can be generated. A guarantee is given that no collection in the executable program can ever contain *null* objects. In addition, when an object $a$ of class $C$ is explicitly deleted (by the expression $a \rightarrow isDeleted()$), then it is also removed from any collection in the system (objects $b$ with a 1-multiplicity association end with value $a$ are deleted along with $a$), so removing the possibility of 'dangling references'.

UML-RSDS has been successfully used for a large number of transformation problems, including industrial case studies. For example, it was used in the comparative case study of [5] and in the transformation tool contest (TTC) in 2010 [6], 2011 [7, 8], 2013 [12] and 2014. This history of use has demonstrated the practical effectiveness of null-free specifications for model transformations.

As an example of UML-RSDS specification, Figure 2 shows the metamodels of a UML to relational database transformation which maps root classes to tables, and attributes to columns of the table of the root class of their class.

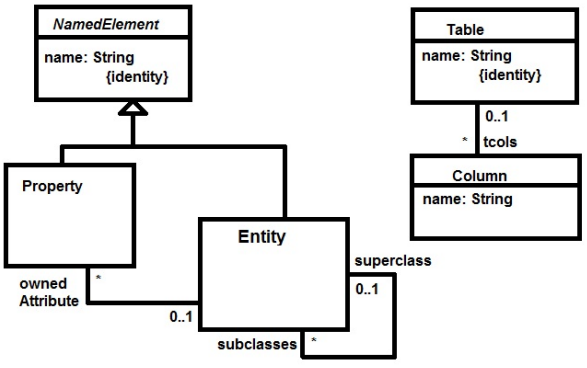


**Fig. 2.** Example of system specification

The assumption $Asm$ is that each class has an ancestor with empty *superclass* set. The transformation use case has two postconditions $C1$, $C2$, both with context *Entity*. $C1$ is:

```
superclass.size = 0  implies  Table->exists( t | t.name = name )
```

This creates a table for each root class. $C2$ is:

```
p : ownedAttribute  and  t = Table[rootClass().name]  implies
        Column->exists( cl | cl.name = p.name  and  cl : t.tcols )
```

This adds, for each attribute $p$ of *self*, a column $cl$ to the table $t$ corresponding to the root class of *self*. *rootClass* is an auxiliary query operation of *Entity* defined as:

```
rootClass( ) : Entity
post:
  ( superclass.size = 0  implies  result = self )  and
  ( superclass.size > 0  implies
                    result = superclass.rootClass()->any(true) )
```

To prove $def(C2)$ requires proof that the call to *rootClass* is well-defined, which follows from $Asm$, and then that $Table[rootClass().name]$ is well-defined, which follows from $C1$.

## 4 Transformation verification

The use of a simplified subset of OCL facilities transformation verification, via the use of semantic mappings from transformation languages into verification formalisms and tools such as B [9], Alloy [1] and Z3 [17]. In the case of UML-RSDS, the classical logic formalisms of B and Z3 are used. For B, class extents $E.allInstances()$ and features $E :: f : T$ are represented by sets $es$ and maps $f : es \rightarrow T'$, respectively, where $T'$ represents $T$, and OCL expressions are mapped into their semantic representation in B. Table 2 summarises some cases of representation of UML in B.

The mapping from UML to B has been implemented in the UML-RSDS tools [11]. This enables a wide range of transformation properties to be verified (in principle) using B proof tools (such as Atelier B or the Rodin Event-B toolkit), which provide both automated and interactive proof support.

The following verification properties of a UML-RSDS transformation specification $\tau$ from a source language $S$ to a target language $T$ can be checked using B:

1. *Syntactic correctness*: if a source model $m$ of $S$ satisfies all the assumptions $Asm$ of $\tau$, then the transformation will produce valid target models $n$ from $m$ which satisfy the language constraints $\Gamma_T$ of $T$.
2. *Invariance*: an invariant $Inv$ is true throughout execution of $\tau$.

| *UML concept* | *Formal semantics/B representation* |
|---|---|
| Entity type $E$ | variable $es$, denoting the set of instances of $E$ |
| Single-valued attribute $att : Typ$ of $E$ | Map $att' : es \rightarrow Typ'$ where $Typ'$ represents $Typ$ |
| Single-valued role $r$ of $E$ with target entity type $E1$ | Map from $es$ to $e1s$: $r' : es \rightarrow e1s$ |
| Unordered many-valued role $r$ of $E$ with target entity type $E1$ | Map from $es$ to $e1s$-sets: $r' : es \rightarrow \mathbb{F}(e1s)$ |
| Ordered many-valued role $r$ of $E$ with target entity type $E1$ | Map from $es$ to $e1s$-sequences: $r' : es \rightarrow \text{seq}(e1s)$ |
| Supertype $E1$ of $E$ | Axiom $es \subseteq e1s$ |

**Table 2.** Representation of UML in B

3. *Semantic preservation*: if a predicate $\varphi$ holds in $m$, then any target model $n$ produced from $m$ by $\tau$ satisfies an interpretation $\chi(\varphi)$ of the formula via a language morphism $\chi$.
4. *Semantic correctness*: that a given implementation for $\tau$ satisfies its specification.
5. *Confluence*: that all result models $n$ produced from a given source model $m$ must be isomorphic.
6. *Termination*: that $\tau$ is guaranteed to terminate if applied to valid source models which satisfy $Asm$.
7. *Model-level semantic preservation*: the internal semantics of source models is preserved, possibly under some interpretation $\zeta$, by $\tau$.

For the UML to relational database example above, a possible invariant (with context *Table*) is that every table corresponds to a root class:

$$Entity \rightarrow exists(e \mid e.name = name \ and \ e.superclass = Set\{\})$$

This can be used to support syntactic correctness proof, i.e., that tables have unique names. Termination, confluence and semantic correctness of the implementation $stat(C1)$; $stat(C2)$ follows from the syntactic form of the constraints (that they can both be implemented by bounded loops).

## 5 Related work

There has been considerable work on the formalisation of OCL and UML for semantic analysis [1, 2, 16]. The complexity of modelling both OCL *null* and *invalid*, together with gaps in the OCL semantic definitions of these values, means that systems such as [2] must make additional assumptions to provide a reasoning framework for full OCL. Instead, we choose to rule out such values completely and to work with classical logic, which has the advantage of unambiguity and the existence of many powerful proof and analysis tools. We agree with

[2] that OCL *invalid* should not be used for modelling, but we would go further and forbid the use of *null* also. By considering that 0..1 multiplicity association ends are collection-valued (of size 0 or 1), a simple and uniform treatment of many data structure properties can be given, without the need to test for *null* values. A similar approach is taken in [15], where association ends are modelled as sequences except for those of multiplicity 1.

## 6 Conclusion

We have provided arguments to justify restrictions upon OCL-style constraint specification in transformation languages, and provided examples to show the practical benefits of such restrictions.

## References


1. K. Anastasakis, B. Bordbar, J. Kuster, *Analysis of Model Transformations via Alloy*, Modevva 2007.
2. A. Brucker, M. Krieger, B. Wolff, *Extending OCL with null-references*, MODELS 2009 Workshops, LNCS 6002, pp. 261–275, 2010.
3. J. Cabot, M. Gogolla, *Object Constraint Language (OCL): a Definitive Guide*, INRIA, 2012.
4. ClearSy, Atelier B, http://www.atelierb.eu, 2012.
5. S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya, P. Van Gorp, *Evaluation of Model Transformation approaches for Model Refactoring*, Science of Computer Programming, 85 (2014), pp. 5–40.
6. K. Lano, S. Kolahdouz-Rahimi, *Migration case study using UML-RSDS*, TTC 2010, Malaga, Spain, July 2010.
7. K. Lano, S. Kolahdouz-Rahimi, *Specification of the "Hello World" case study*, TTC 2011.
8. K. Lano, S. Kolahdouz-Rahimi, *Specification of the GMF migration case study*, TTC 2011.
9. K. Lano, S. Kolahdouz-Rahimi, T. Clark, *Comparing verification techniques for model transformations*, Modevva workshop, MODELS 2012.
10. K. Lano, S. Kolahdouz-Rahimi, *Constraint-based specification of model transformations*, Journal of Systems and Software, vol. 88, no. 2, February 2013, pp. 412–436.
11. K. Lano, *The UML-RSDS manual*, http://www.dcs.kcl.ac.uk/staff/kcl/umlrsds.pdf, 2013.
12. K. Lano, S. Kolahdouz-Rahimi, K. Maroukian, *Solving the Petri-Nets to Statecharts Transformation Case with UML-RSDS*, TTC 2013.
13. OMG, *Object Constraint Language 2.3.1 Specification*, 2012.
14. OMG, *QVT Specification*, Version 1.1, 2011.
15. I. Poernomo, *A type theoretic semantics for the OMG Meta-Object Facility*, PALAB, Dept. of Informatics, King's College London, 2009.
16. M. Soeken, R. Wille, R. Drechsler, *Encoding OCL data types for SAT-based verification of UML/OCL models*, University of Bremen, 2012.
17. Z3 Theorem Prover, http://research.microsoft.com/en-us/um/redmond/projects/z3/, 2012.