Null Considered Harmful (for Transformation Verification)

Kevin Lano

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

OCL null and invalid

- OCL includes explicit *null* and *invalid* values: these values complicate logic of OCL and transformation languages that use OCL, hindering verification

$$x/y < 1 \textit{ or } (x/y).\textit{oclIsUndefined}()$$

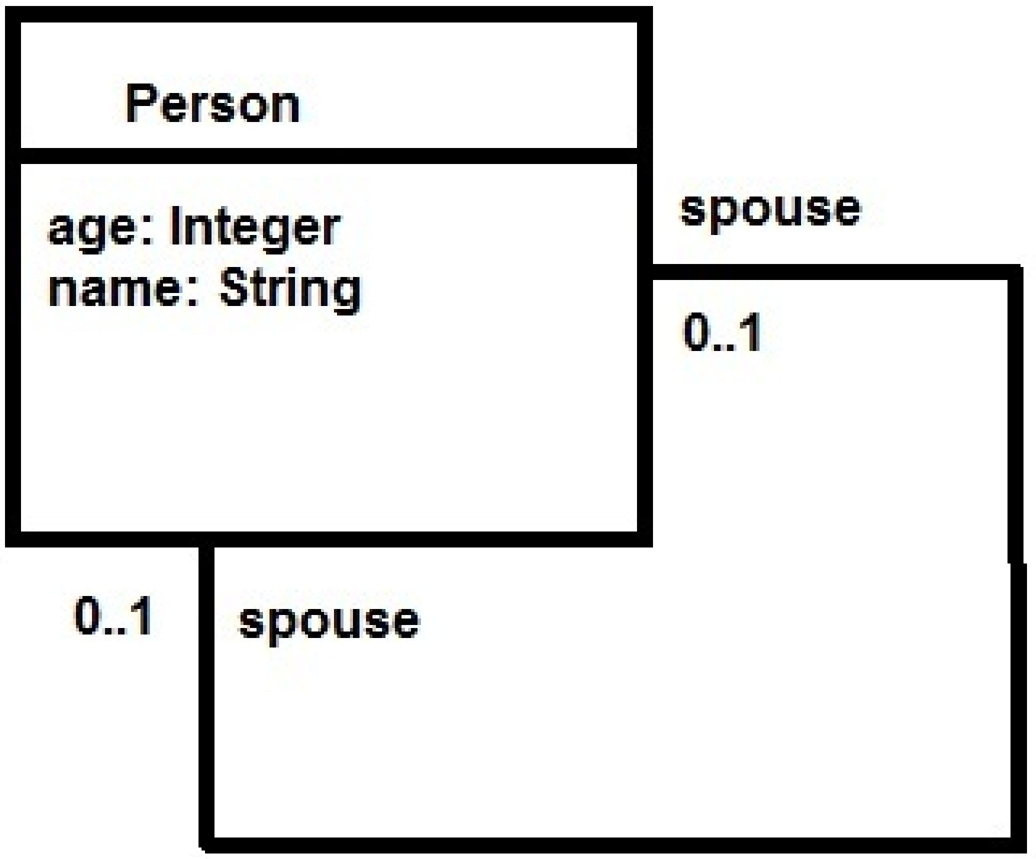
- We define restricted OCL which avoids explicit null and undefined values
- Give examples of verification techniques for transformation language, UML-RSDS, based on this approach.

Problems with null and invalid

OCL type *OclVoid* has single instance, *null*. *OclVoid* is subtype of all other OCL types (except *OclInvalid*, which contains *invalid*), so each type, including Boolean and Integer, has *null* element. *null* represents absence of value, in contrast to *invalid*, which represents an invalid evaluation.

null used for value of 0..1 association ends: if a person has no spouse, this is expressed by $spouse = null$ – specifications involving such ends need this test before evaluating features of the end.

Null values can also be returned by operations – can lead to poor specification styles.



Example of optional association ends

Problems with null and invalid

Complicate logic used for reasoning. OCL truth tables for logical operators incorporate cases for *null*, e.g.:

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

invalid or true is *true* (different to short-circuit evaluation in Java, strict evaluation in ATL, etc).

To use classical 2-valued logic verification tools (e.g., B or Z3), must model 3 or 4-valued OCL logic in tool. Can be inefficient and difficult for specifiers and analysts to use.

Problems with null and invalid

- Unresolved + unclear issues with semantics of *null* and *invalid*, in OCL (2.3.1, 2.4) standard, and in MT languages
- E.g.: $null.oclIsKindOf(C) = invalid$ but $null.oclAsType(C) = null$ (contradicts general definition of *oclIsKindOf* + clause A.30)
- Can *null* elements be in collections? (OCL: yes, QVT-O: no)
- OCL used imperatively in some MT languages
- *invalid* sometimes used for empty 0..1 association ends, instead of *null*
- Lack of clear semantics, and variability of definitions indicates use of explicit *null* and *undefined* elements is inappropriate if formal verification of MT required.

Restricted OCL

We propose alternative approach for use of OCL/constraint languages in MT specifications, in accordance with following principles:

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

Principles facilitate MT analysis and verification, ensure expressions can be given consistent meanings in (i) MT language; (ii) formal languages used for analyses, and (iii) implementation programming languages used.

Restricted OCL

- Make logic classical by excluding *null* and *invalid* from expression values
- Side-effects only permitted for calls $obj.op(e)$ of update (non-query) operations op. Expressions should not have side-effects when used as conditions or pure values
- Same classical logic for logical connectives both in OCL and in formal and programming languages
- Collection operators classically defined as set comprehensions, etc., not via iterators.

Definedness definition

Definedness function $\mathit{def} : \mathit{Exp}(L) \rightarrow \mathit{Exp}(L)$. Expression determinacy $\mathit{det} : \mathit{Exp}(L) \rightarrow \mathit{Exp}(L)$ similarly defined.

Constraint expression e	*def(e)*
a/b	$b \neq 0$ *and* $def(a)$ *and* $def(b)$
$e.f$ Data feature application	$def(e)$ *and* $E.allInstances() \rightarrow includes(e)$ E is declared classifier of e
$s \rightarrow at(ind)$ sequence s	$ind > 0$ *and* $ind \leq s \rightarrow size()$ *and* $def(s)$ *and* $def(ind)$
$E[v]$ entity type E with identity attribute id, v single-valued	$E.id \rightarrow includes(v)$ *and* $def(v)$
A *and* B A *implies* B	$def(A)$ *and* $def(B)$ $def(A)$ *and* (A *implies* $def(B)$)
$str.toInteger()$ $str.toReal()$ $obj.oclAsType(E)$	$def(str)$ *and* $str.isInteger()$ $def(str)$ *and* $str.isReal()$ $def(obj)$ *and* $obj.oclIsKindOf(E)$

Restricted OCL

Optional association ends are treated as *collections* of size 0 or 1. Test $spouse {\rightarrow} isEmpty()$ 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, no need for *null* and *invalid*.
- Simple to evolve a model using a 0..1 multiplicity association end to one with 0..n or * multiplicity, and vice-versa.
- Collections cannot contain *null* elements, simplifying their processing, both in OCL and in programs synthesised from the OCL.

Transformation specification

Restricted OCL can be used to define transformations, by pre and postconditions. Transformation τ has assumptions Asm on starting state of input models and postconditions $Cons$.

Typical postcondition Cn form is

$$Ante \ implies \ Succ$$

on context entity type ST. Asserts for each instance *self* of ST which satisfies $Ante$, $Succ$ also holds.

Definedness $def(Cn)$, determinacy $det(Cn)$ should follow from context ST of Cn, Asm and preceding postconditions.

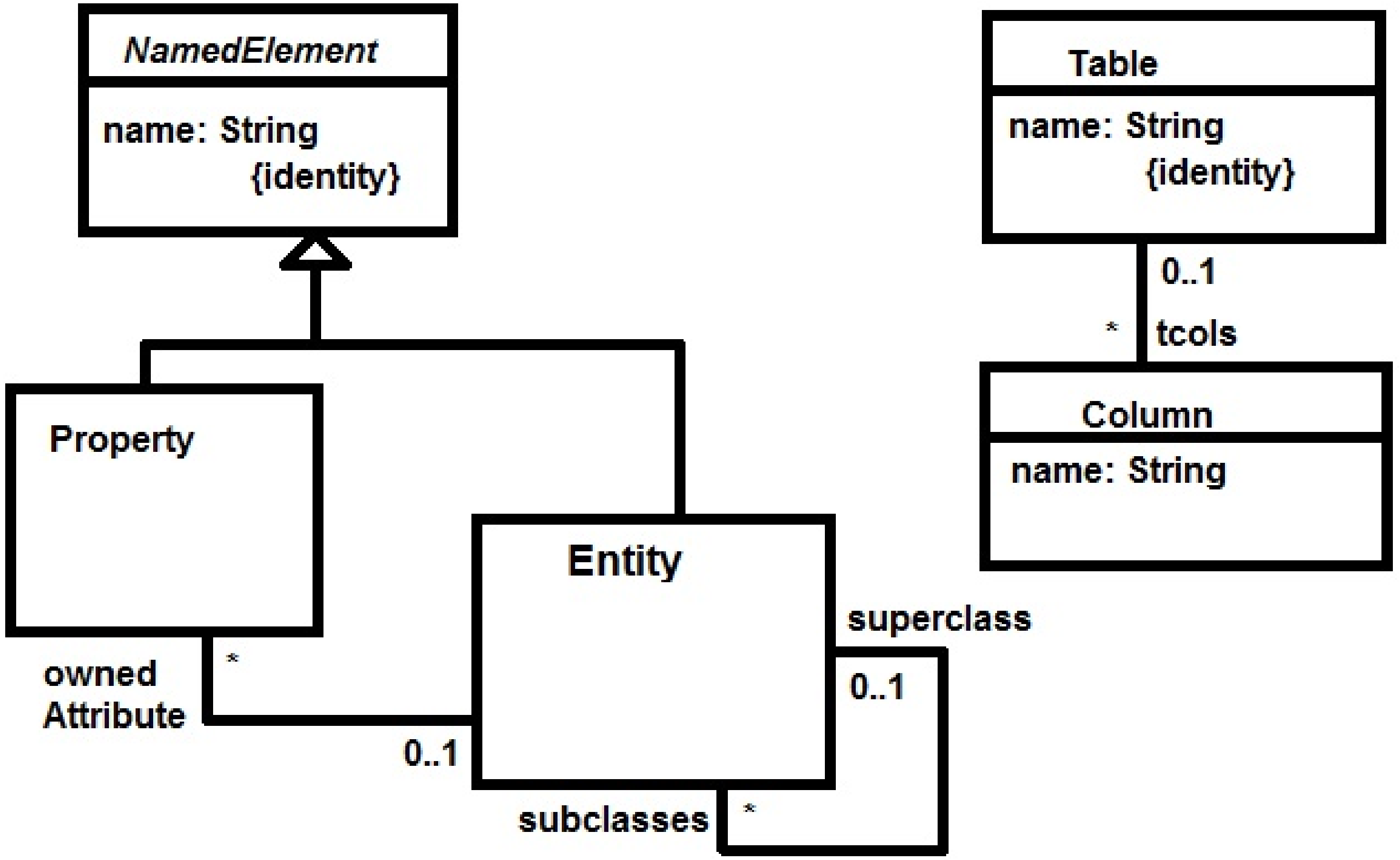
$Succ$ should have an imperative denotation $stat(Succ)$.

Transformation specification

Example: UML to relational database transformation which maps root classes to tables, attributes to columns of table of the root class of their class.

Assumption Asm is that each class has an ancestor with empty *superclass* set.

Transformation has postconditions $C1$, $C2$, with context $Entity$.



Example of system specification

$C1$ 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 table t corresponding to root class of *self*.

rootClass is query operation of *Entity*:

```
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 call to $rootClass$ is well-defined, which follows from Asm, and that $Table[rootClass().name]$ is well-defined, which follows from $C1$.

Transformation verification

Restricted OCL facilitates transformation verification, via semantic mappings from transformation languages into verification formalisms and tools such as B, Alloy and Z3.

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

Transformation verification

For UML to relational database example, possible transformation 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 implementation $stat(C1);\; stat(C2)$ follows from syntactic form of the constraints (both can be implemented by bounded loops).

Conclusion

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

By restricting to OCL subset with classical logic semantics, can provide verification for wide range of MTLs + using any verification tool based on classical logic.