

# OCL2FOL<sup>+</sup>: Coping with Undefinedness

Carolina Dania<sup>1,2</sup> and Manuel Clavel<sup>1,2</sup>

<sup>1</sup> IMDEA Software Institute, Madrid, Spain  
[carolina.dania,manuel.clavel]@imdea.org

<sup>2</sup> Universidad Complutense de Madrid, Madrid, Spain  
clavel@sip.ucm.es

**Abstract.** At present, the OCL language includes two constants, `null` and `invalid`, to represent undefinedness. This effectively turns OCL into a four-valued logic. It makes also problematic its mapping to first-order logic and, as a consequence, hinders the use of first-order automated-reasoning tools for OCL reasoning. We address this problem and propose a solution, grounded on the same principles underlying OCL2FOL, in order to cope with undefinedness in OCL.

## 1 Introduction

In the past decade, there has been a plethora of proposals to map OCL [12] into different formalisms for which reasoning tools may be readily available. By choosing a particular formalism each proposal commits to a different trade off between expressiveness and termination, automation, or completeness (see [13, 6] and references). In particular, most of these proposals have not considered the OCL language parts that deal with *undefinedness* or generate it: basically, the constants `null` and `invalid`,<sup>3</sup> the type-casting operator, the division operator, and the operators `any`, `oclIsUndefined` and `oclIsInvalid`. By doing so, these proposals try to turn OCL into a two-valued logic, so that it can be more easily mapped to different formalisms for which reasoning tools already exist. Notable exceptions to this trend are [4, 5, 3, 11].

In [7] we proposed a mapping from OCL to first-order logic, and we implemented it in a tool called OCL2FOL. This mapping supports the direct use of SMT solvers (e.g., Z3 [8] or Yices [9]) to check the satisfiability of OCL expressions, but under specific restrictions both on the expressions that were allowed and on the instances of data models that were checked.<sup>4</sup> Not surprisingly, some of these restrictions were imposed to avoid the problem of coping with undefinedness in OCL. In particular, i) OCL2FOL only considers instances where all attributes are defined, and ii) it does not allow expressions containing operators that can generate undefined values. However, after experimenting with

---

<sup>3</sup> Intuitively, the constant `null` represents unknown/undefined values, whereas the constant `invalid` represents error/exceptions.

<sup>4</sup> Class diagrams and object diagrams are prototypical examples, respectively, of data models and of instances of data models.

OCL2FOL, we have come to realize that these restrictions impose severe limitations on the applicability of the tool (and we conjecture that similar limitations will apply as well to other methodologies that are not prepared to deal with undefinedness).

In this paper we propose OCL2FOL<sup>+</sup> which, although grounded on the same principles underlying OCL2FOL, is designed to overcome the limitations of the latter with regard to undefinedness in OCL.<sup>5</sup> Let us explain the key difference in a nutshell. Under the aforementioned restrictions i) and ii), a Boolean OCL expression can only evaluate to either `true` or `false`. Thus, to reason in this context using Boolean OCL expressions it is sufficient to define a mapping that formalizes when a Boolean OCL expressions evaluates to `true`. This is precisely what we did in OCL2FOL. However, in the presence of undefinedness, Boolean OCL expressions can also evaluate to `null` or `invalid`. To cope with this fact, we now define in OCL2FOL<sup>+</sup> four different mappings which formalize, respectively, when a Boolean OCL expression evaluates to `true`, when to `false`, when to `null`, and when to `invalid`.

*Organization.* First, in Section 2, we recall the basic principles underlying the mapping from OCL to first-order logic which is implemented in OCL2FOL. Then, in Section 3, we discuss the key ideas behind the four mappings which are implemented in OCL2FOL<sup>+</sup> for coping with undefinedness in OCL. Next, in Section 4, we report on the status of OCL2FOL<sup>+</sup>, and on some experiments that we have carried out with this tool. Finally, we draw conclusions in Section 5.

## 2 OCL2FOL

In [7] we define a mapping, called OCL2FOL, from a subset of OCL<sup>6</sup> into first-order logic (FOL). For the sake of space, we will refer here to this mapping as O2F\*. The key property of O2F\* is the following: Let  $M$  be a data model, let  $I$  be an instance of  $M$  (where all attributes values are defined) and let  $expr$  be a Boolean OCL expression (within the O2F\*'s range), whose context is also  $M$ . Then,

$$\text{Eval}(expr, I) = \text{true} \iff \text{O2F}^{\text{env}}(M, I, expr) \models \text{O2F}^*(expr) \quad (1)$$

where  $\text{Eval}(expr, I)$  is the value returned by the evaluation of the expression  $expr$  for the instance  $I$ , and  $\text{O2F}^{\text{env}}(M, I, expr)$  is the union of the set of formulas resulting from calling three auxiliary mappings, namely,  $\text{O2F}^{\text{data}}$ ,  $\text{O2F}^{\text{inst}}$ , and  $\text{O2F}^{\text{def}}$ , on, respectively, the data model  $M$ , the instance  $I$ , and the expression  $expr$ . That is,

$$\text{O2F}^{\text{env}}(M, I, expr) = \text{O2F}^{\text{data}}(M) \cup \text{O2F}^{\text{inst}}(I) \cup \text{O2F}^{\text{def}}(expr)$$

<sup>5</sup> However, other limitations, like the inability to deal with the operator `size` and `count`, or with collection-types different from `Set` do still apply to OCL2FOL<sup>+</sup>.

<sup>6</sup> This subset includes arbitrary, possibly nested `forAll`, `exists`, `select`, `reject`, and `collect` iterator expressions. However, `collect`-expressions must be followed by `asSet`. See [7, 2] for further details.

Next, we briefly discuss these three auxiliary mappings, and then the key mapping  $\text{O2F}^*$ . But before, consider the following remark, whose correctness is based on the correctness of (1) and on the assumption that the evaluation of  $expr$  cannot possible result in `null` or `invalid`.

*Remark 1.* Let  $M$  be a data model, let  $I$  be an instance of  $M$  (where all attributes values are defined), and let  $expr$  be a Boolean OCL expression (within the  $\text{O2F}^*$ 's range), whose context is also  $M$ . Then,

$$\text{Eval}(expr, I) = \text{false} \iff \text{O2F}^{\text{env}}(M, I, expr) \not\equiv \text{O2F}^*(expr)$$

### $\text{O2F}^{\text{data}}(M)$ : Mapping data models $M$

To simplify the presentation, we do not consider here class inheritance, multi-valued attributes, or association-ends with multiplicities different from `*`.

Classes are represented by predicates. In particular, for every two classes  $C$  and  $C'$  in  $M$ ,  $C \neq C'$ , the set  $\text{O2F}^{\text{data}}(M)$  includes the following formula:

- $\forall(x)(\neg(C(x) \wedge C'(x)))$ .

Then, attributes are represented as functions. Finally, association-ends are represented as binary predicates. In particular, for every association between two classes  $C$  and  $C'$ , with association-ends  $assoc$  and  $assoc'$ , the set  $\text{O2F}^{\text{data}}(M)$  includes the following formulas:

- $\forall(x, y)(assoc(x, y) \Leftrightarrow assoc'(y, x))$
- $\forall(x, y)(assoc(x, y) \Rightarrow C'(y)) \wedge \forall(x, y)(assoc(x, y) \Rightarrow C(x))$

### $\text{O2F}^{\text{inst}}(I)$ : Mapping instances $I$

Objects are represented by constants. In particular, for every object  $o$  in  $I$  of a class  $C$ , the set  $\text{O2F}^{\text{inst}}(I)$  includes the following formula:

- $C(o)$

Then, attributes values are represented by equalities, which all-together define the corresponding functions. In particular, for every object  $o$  in  $I$ , if  $v$  is the value of its attribute  $attr$ , then the set  $\text{O2F}^{\text{inst}}(I)$  includes the following equality:

- $attr(o) = v$ .

Finally, links between objects are represented by formulas, which all-together define the corresponding predicates. In particular, for every two objects  $o$  and  $o'$  in  $I$ , if  $o'$  is linked to  $o$ 's association-end  $assoc$ , then the set  $\text{O2F}^{\text{inst}}(M)$  includes the following formula:

- $assoc(o, o')$ .

### $O2F^{def}(expr)$ : Mapping definitions in expressions *expr*

The OCL language includes operators (e.g., `collect` or `select`, but also `union` or `including`) that define new collections, without naming them, using a certain *property*. We represent these new collections by new predicates. In particular, for every sub-expression of *expr* which defines a new collection, the set  $O2F^{def}(expr)$  includes the formula that formalizes that for every element for which the new predicate holds it also holds the property which defines the new collection. Example 1 below illustrates, among other things, this essential feature of our mapping from OCL to first-order logic.

There are also operators in OCL (e.g., `max` or `any`) that refer to objects, without naming it, using also a certain *property*. We represent these objects by new constants. In particular, for every sub-expression of *expr* that refers, without naming it, to an object using a property, then set  $O2F^{def}(expr)$  includes the formula that formalizes that the aforementioned property holds for the element referred to by the new constant.

### $O2F^*(expr)$ . Mapping expressions (but not aware of null or invalid)

The map  $O2F^*$  is defined *recursively* over the structure of OCL expressions, according to the following principles: First, each expression of the form *C.allInstances* is translated by a predicate formula whose predicate is the one representing the class *C*. Second, each expression of type **Boolean** whose root symbol (the one at the root of the tree representing the expression) is `not`, `and`, `or`, `implies`, `xor`, `=`, `>`, `<`, `≤`, `≥`, `forAll`, `exists`, `one`, `isUnique`, `isEmpty`, `notEmpty`, `includes`, `excludes`, `includesAll`, `excludesAll` is translated into a formula which mirror its logical structure. Third, each expression of type **Integer** whose root symbol is `+`, `-`, `*` is translated by the corresponding functional expression. Fourth, each expression of type **Set** whose root symbol is `select`, `reject`, `including`, `excluding`, `union`, `intersection`, `symmetricDifference`, `collect` (followed by `asSet`) is translated by a predicate formula, whose predicate's definition is generated using  $O2F^{def}$ . Finally, each expression of type **Integer** whose root symbol is `max` or `min` (over collections) is translated by a constant, whose definition is generated using  $O2F^{def}$ .

The recursive definition of  $O2F^*$  is rather technical, and we shall refer to [7, 2] for further details. These technicalities are mainly due to the fact that, in order to capture the semantics of a number of OCL operators (including the iterator operators, but also others) we introduce *fresh* logical variables, which are then passed on to the subsequent recursive calls and are used when translating the arguments of these operators. This and other features of the mapping from OCL to first-order logic which is implemented in OCL2FOL are illustrated by the following example.

*Example 1.* Consider a simple data model **Persons** containing just one class **Person**, with one attribute `age` of type **Integer** and a self-association whose associations are `friendsOf` (*x* is a friend of *y*) and `friendTo` (*y* is considered a friend by *x*).

Let  $expr$  be  $\text{Person.allInstances()}\rightarrow\text{notEmpty()}$ . Then,  $\text{O2F}^*(expr)$  is the following formula:

$$\exists(x)(\text{Person}(x))$$

where  $\text{Person}$  is the predicate that represents the class `Person` and  $x$  is a logical variable introduced as part of the mapping of the operator `notEmpty`.

Now, let  $expr$  be  $\text{Person.allInstances()}\rightarrow\text{exists}(p|p.\text{age} \leq 18)$ . Then,  $\text{O2F}^*(expr)$  is the following formula:

$$\exists(x)(\text{Person}(x) \wedge \text{age}(x) \leq 18)$$

where  $\text{age}$  is the function that represents the attribute `age` and  $x$  is a logical variable introduced as part of the mapping of the operator `exists`.

Also, let  $expr$  be  $\text{Person.allInstances()}\rightarrow\text{select}(p|p.\text{age} > 18) \rightarrow\text{isEmpty()}$ . Then,  $\text{O2F}^*(expr)$  is the following formula:

$$\forall(x)(\neg\text{Select\#1}(x))$$

where  $x$  is a logical variable introduced as part of the mapping of the operator `isEmpty` and  $\text{Select\#1}$  is the predicate that represents the collection defined by the expression  $\text{Person.allInstances()}\rightarrow\text{select}(p|p.\text{age} > 18)$ . The predicate  $\text{Select\#1}$  is, in turn, defined, using the mapping  $\text{O2F}^{\text{def}}$ , by the following formula:

$$\forall(x)(\text{Select\#1}(x) \Leftrightarrow (\text{Person}(x) \wedge \text{age}(x) > 18))$$

Finally, let  $expr$  be  $\text{Person.allInstances()}\rightarrow\text{collect}(p|p.\text{friendsOf})\rightarrow\text{asSet()}\rightarrow\text{notEmpty()}$ . Then,  $\text{O2F}^*(expr)$  is the following formula:

$$\exists(x)(\text{Collect\#1}(x))$$

where  $x$  is a logical variable introduced as part of the mapping of the operator `notEmpty` and  $\text{Collect\#1}$  is the predicate that represents the collection defined by the expression  $\text{Person.allInstances()}\rightarrow\text{collect}(p|p.\text{friendsOf})\rightarrow\text{asSet()}$ . The predicate  $\text{Collect\#1}$  is, in turn, defined, using the mapping  $\text{O2F}^{\text{def}}$ , by the following formula:

$$\forall(x)(\text{Collect\#1}(x) \Leftrightarrow \exists(y)(\text{Person}(y) \wedge \text{friendsOf}(y, x)))$$

where  $\text{friendsOf}$  is the predicate that represents the association-end `friendsOf`.

### 3 OCL2FOL<sup>+</sup>: Coping with undefinedness

In a nutshell, the problem of undefinedness in OCL when trying to use existing first-order logic reasoning tools for OCL reasoning is the problem of mapping a four-valued logic (`true`, `false`, `null`, and `invalid`) into a two-valued logic (*true* and *false*). In general, when evaluating a Boolean OCL expression for an instance of a data model, the result can be either `true`, `false`, `null` or `invalid`. However, when interpreting a formula in a first-order model, the result can only be *true* or *false*. In other words, in the presence of undefinedness, Remark 1 does not hold.

To cope with undefinedness while at the same time being able to use SMT solvers, as we did in OCL2FOL, to automatically carry out OCL reasoning, our solution consists on defining four different mappings, namely,  $O2F^{\text{true}}$ ,  $O2F^{\text{false}}$ ,  $O2F^{\text{null}}$ , and  $O2F^{\text{invalid}}$ , such that the following holds: Let  $M$  be a data model, let  $I$  be an instance of  $M$  (where now not all attributes values need to be defined), and let  $expr$  be a Boolean OCL expression whose context is  $M$  (within the  $O2F^*$ 's range, extended now to include expressions built with the type-casting operators, the division operator, and the operators `any`, `oclIsUndefined`, and `oclIsInvalid`). Then,

$$\begin{aligned} \text{Eval}(expr, I) = \text{true} &\iff O2F^{\text{env}+}(M, I, expr) \models O2F^{\text{true}}(expr) \\ \text{Eval}(expr, I) = \text{false} &\iff O2F^{\text{env}+}(M, I, expr) \models O2F^{\text{false}}(expr) \\ \text{Eval}(expr, I) = \text{null} &\iff O2F^{\text{env}+}(M, I, expr) \models O2F^{\text{null}}(expr) \\ \text{Eval}(expr, I) = \text{invalid} &\iff O2F^{\text{env}+}(M, I, expr) \models O2F^{\text{invalid}}(expr) \end{aligned}$$

where  $\text{Eval}(expr, I)$  is the value returned by the evaluation of the expression  $expr$  for the instance  $I$ , and  $O2F^{\text{env}+}(M, I, expr)$  is the union of the set of formulas resulting from calling three auxiliary mappings, namely,  $O2F^{\text{data}+}$ ,  $O2F^{\text{inst}+}$ , and  $O2F^{\text{def}+}$ , on, respectively, the data model  $M$ , the instance  $I$ , and the expression  $expr$ . As their names suggest, these three mappings are extensions of the mappings  $O2F^{\text{data}}$ ,  $O2F^{\text{inst}}$ , and  $O2F^{\text{def}}$ . That is,

$$O2F^{\text{env}+}(M, I, expr) = O2F^{\text{data}+}(M) \cup O2F^{\text{inst}+}(I) \cup O2F^{\text{def}+}(expr)$$

Next, we briefly discuss the content of these extensions, and afterwards the four key mappings  $O2F^{\text{true}}$ ,  $O2F^{\text{false}}$ ,  $O2F^{\text{null}}$ , and  $O2F^{\text{invalid}}$ .

### **$O2F^{\text{data}+}(M)$ : Mapping data models $M$**

The values `null` or `invalid` are represented, respectively, by the constants *null* and *invalid*. Moreover, we introduce two unary predicates *IsNull* and *IsInvalid*, to represent when an element is `null` or `invalid`, and we add to the set  $O2F^{\text{data}+}(M)$  the following formula:

- $IsNull(\text{null}) \wedge \neg IsInvalid(\text{null}) \wedge \neg IsNull(\text{invalid}) \wedge IsInvalid(\text{invalid})$ .

Objects are neither `null` nor `invalid`.<sup>7</sup> Thus, for every class  $C$  in  $M$ , the set  $O2F^{\text{data}+}(M)$  contains the following additional formula:

- $\forall(x)(C(x) \Rightarrow \neg(IsNull(x) \vee IsInvalid(x)))$ .

<sup>7</sup> At the model level, objects are instances of (UML) classes. Thus, they cannot be `null` nor `invalid`. At the linguistic level, however, the constants *null* and *invalid* are instances of every (OCL) class type.

### **O2F<sup>inst+</sup>(I): Mapping instances I**

Attributes values may be undefined. Thus, for every object  $o$  in  $I$ , if the value of its attribute  $attr$  is undefined, then the set  $O2F^{\text{inst}+}(I)$  includes the following equality:

- $attr(o) = null$ .

On the other hand, attributes values may be defined. Thus, for every object  $o$  in  $I$ , if  $v$  is the (defined) value of its attribute  $attr$ , then the set  $O2F^{\text{inst}+}(I)$  includes the following formula:

- $\neg(IsNull(v) \vee IsInvalid(v))$ .

### **O2F<sup>def+</sup>(expr): Mapping definitions in expressions expr**

Literals of primitive types are also neither null nor invalid. Thus, for every literal  $l$  in the given expression  $expr$ , the set  $O2F^{\text{def}+}(expr)$  contains the following additional formula:

- $\neg(IsNull(l) \vee IsInvalid(l))$ .

### **Mapping expressions (aware of null and invalid)**

For the sake of space limitation, we shall not provide here the full recursive definitions of our four mappings  $O2F^{\text{true}}$ ,  $O2F^{\text{false}}$ ,  $O2F^{\text{null}}$ ,  $O2F^{\text{invalid}}$ . Instead, we illustrate the main ideas using examples. In particular, since the new mappings follow the same principles than our original mapping  $O2F^*$ , we use as examples the same expressions that we used in Example 1.

According to the OCL's semantics, if a collection contains *invalid*, the collection itself evaluates to *invalid*. Furthermore, for every operator, except of course `oclIsInvalid` and `oclIsUndefined`, if *invalid* happens to be passed as one of its arguments, the result is *invalid*. We now take these facts into account when mapping operators on collection, as shown in the following example. Notice, in particular, that we do not “reduce” a collection containing *invalid* to *invalid*, but instead we “keep” it in that collection. Then, when formalizing the semantics of operators on collections, we “check” if the given collections may contain *invalid*, and proceed accordingly.

*Example 2.* Let  $expr$  be `Person.allInstances()→notEmpty()`.

$$O2F^{\text{true}}(expr) = \exists(x)(Person(x)) \wedge \forall(x)(Person(x) \Rightarrow \neg IsInvalid(x))$$

$$O2F^{\text{false}}(expr) = \forall(x)(\neg Person(x)) \wedge \forall(x)(Person(x) \Rightarrow \neg IsInvalid(x))$$

$$O2F^{\text{null}}(expr) = \perp$$

$$O2F^{\text{invalid}}(expr) = \exists(x)(Person(x) \wedge IsInvalid(x))$$

Notice that, in the logical context provided by  $O2F^{env+}$ , some of the above expressions can be simplified. For example,  $\forall(x)(Person(x) \Rightarrow \neg IsInvalid(x))$  can be reduced to  $\top$ , since objects are neither `null` nor `invalid`; for the same reason,  $\exists(x)(Person(x) \wedge IsInvalid(x))$  can be reduced to  $\perp$ . However, for the sake of clarity, we will not perform such optimizations in our examples.

Now, if one of the elements in a disjunction is `null`, and the other element is neither `true` nor `invalid`, then the disjunction itself evaluates to `null`. Similarly, if one the elements in a disjunction is `invalid`, and the other element is not `true`, then the disjunction itself evaluates to `invalid`. Now, in the case of the operator  $\leq$ , if both arguments are `null`, then  $\leq$  evaluates to `true`. But if only one of the arguments, but not the other, is `null`, then  $\leq$  evaluates to `invalid`. And, if at least one element is `invalid`, or one argument is `null`, but not the other, then  $\leq$  evaluates to `invalid`. We take these facts into account when mapping the iterator `exists` over collections and the operator  $\leq$  between integers, as shown in the following examples.

*Example 3.* Let `expr` be `Person.allInstances() → exists(p|p.age ≤ 18)`.

$$\begin{aligned}
O2F^{true}(expr) &= \\
&\exists(x)(Person(x) \wedge O2F^{true}(x.age \leq 18)) \wedge \forall(x)(Person(x) \Rightarrow \neg IsInvalid(x)) \\
O2F^{false}(expr) &= \\
&\forall(x)(Person(x) \Rightarrow O2F^{false}(x.age \leq 18)) \wedge \forall(x)(Person(x) \Rightarrow \neg IsInvalid(x)) \\
O2F^{null}(expr) &= \\
&\exists(x)(Person(x) \wedge O2F^{null}(x.age \leq 18)) \\
&\wedge \forall(x)(Person(x) \Rightarrow (O2F^{false}(x.age \leq 18) \vee O2F^{null}(x.age \leq 18))) \\
&\wedge \forall(x)(Person(x) \Rightarrow \neg IsInvalid(x)) \\
O2F^{inval}(expr) &= \\
&(\exists(x)(Person(x) \wedge O2F^{inval}(x.age \leq 18)) \\
&\quad \wedge \forall(x)(Person(x) \Rightarrow (O2F^{false}(x.age \leq 18) \vee O2F^{null}(x.age \leq 18) \\
&\quad \vee O2F^{inval}(x.age \leq 18)))) \\
&\vee \exists(x)(Person(x) \wedge IsInvalid(x))
\end{aligned}$$

where, if `expr` is `p.age ≤ 18`, then

$$\begin{aligned}
O2F^{true}(expr) &= \\
&(O2F^{null}(p.age) \wedge O2F^{null}(18)) \vee \\
&(age(p) \leq 18 \\
&\quad \wedge \neg(O2F^{null}(p.age) \vee O2F^{null}(18) \vee O2F^{inval}(p.age) \vee O2F^{inval}(18))) \\
O2F^{false}(expr) &= \\
&\neg(age(p) \leq 18) \\
&\quad \wedge \neg(O2F^{null}(p.age) \vee O2F^{null}(18) \vee O2F^{inval}(p.age) \vee O2F^{inval}(18)) \\
O2F^{null}(expr) &= \perp \\
O2F^{inval}(expr) &= \\
&O2F^{inval}(p.age) \vee O2F^{inval}(18) \vee (O2F^{null}(p.age) \wedge \neg(O2F^{null}(18))) \\
&\vee (O2F^{null}(18) \wedge \neg(O2F^{null}(p.age)))
\end{aligned}$$



Next, recall that a collection resulting from the `select`-iterator contains every element of the source-collection for which the body evaluates to `true`. However, we now take into account that, if the source-collection contains `invalid` (and the same applies to the other iterator operators), then the resulting collection will also contain `invalid`, as shown in the following example.

*Example 4.* Let  $expr$  be `Person.allInstances()→select(p|p.age > 18)→isEmpty()`.

$$O2F^{true}(expr) = \forall(x)(\neg Select\#1(x) \wedge \forall(x)(Select\#1(x) \Rightarrow \neg IsInvalid(x)))$$

$$O2F^{false}(expr) = \exists(x)(Select\#1(x) \wedge \forall(x)(Select\#1(x) \Rightarrow \neg IsInvalid(x)))$$

$$O2F^{null}(expr) = \perp$$

$$O2F^{invalid}(expr) = \exists(x)(Select\#1(x) \wedge IsInvalid(x))$$

where  $Select\#1$  is the predicate that represents the collection defined by the expression `Person.allInstances()→select(p|p.age > 18)`. The predicate  $Select\#1$  is, in turn, defined by the following formula:

$$\forall(x)(Select\#1(x) \Leftrightarrow (Person(x) \wedge (O2F^{true}(x.age > 18) \vee IsInvalid(x))))$$

Finally, recall that `oclIsUndefined` returns `true` when its argument is either `null` or `invalid`, and `false` otherwise. Also, recall that the `any`-iterator returns `null` if no element in the source-collection satisfies the given property. As expected, if the source-collection contains `invalid`, the `any`-iterator also returns `invalid`. We now take into account all these three facts, as shown in the following example:

*Example 5.* Let  $expr$  be `Person.allInstances()→any(p|p.age > 16).oclIsUndefined()`.

$$O2F^{true}(expr) = IsNull(Any\#1) \vee IsInvalid(Any\#1)$$

$$O2F^{false}(expr) = \neg(IsNull(Any\#1) \vee IsInvalid(Any\#1))$$

$$O2F^{null}(expr) = \perp$$

$$O2F^{invalid}(expr) = \perp$$

where  $Any\#1$  is the constant that represents the element defined by the expression `Person.allInstances()→any(p|p.age > 16)`. The constant  $Any\#1$  is, in turn, defined by the following formulas:

$$\begin{aligned} \neg(IsNull(Any\#1) \vee IsInvalid(Any\#1)) \Rightarrow \\ \exists(x)(Person(x) \wedge O2F^{true}(x.age > 16) \wedge Any\#1 = x) \end{aligned}$$

$$\begin{aligned} IsNull(Any\#1) \Leftrightarrow \\ (\forall(x)(Person(x) \Rightarrow (O2F^{false}(x.age > 16) \vee O2F^{null}(x.age > 16) \\ \vee O2F^{invalid}(x.age > 16)))) \end{aligned}$$

$$\wedge \forall(x)(Person(x) \Rightarrow \neg IsInvalid(x))$$

$$IsInvalid(Any\#1) \Leftrightarrow \exists(x)(Person(x) \wedge IsInvalid(x))$$

1. `Person.allInstances() ->forall(p|p.age>18)`
2. `Person.allInstances() ->exists(p|p.age<=18)`
3. `Person.allInstances() ->exists(p|p.age.ocllsUndefined())`
4. `Person.allInstances() ->exists(p|p.ocllsUndefined())`
5. `Person.allInstances() ->forall(p|p.ocllsUndefined())`
6. `Person.allInstances() ->notEmpty()`
7. `Person.allInstances() ->collect(p|p.age) ->asSet() ->exists(a|a.ocllsUndefined())`
8. `Person.allInstances() ->any(p|p.age>16).ocllsUndefined()`
9. `Person.allInstances() ->any(p|p.age>16).age.ocllsInvalid()`
10. `not(Person.allInstances() ->any(p|p.age<16).age.ocllsInvalid())`

Fig. 1. OCL2FOL<sup>+</sup> case study: sample formulas

## 4 Tool support and case study

As part of this work, we have implemented our four mappings in a tool, called OCL2FOL<sup>+</sup> [2]. OCL2FOL<sup>+</sup> takes as input a data model  $M$  (the *context*), a set of Boolean OCL expressions  $exp_1, \dots, exp_n$  (the *hypothesis*), a Boolean OCL expression  $exp_{n+1}$  (the *assertion*), and one of the following keywords (the *type*): **true**, **false**, **null**, **inval**. From the given input, OCL2FOL<sup>+</sup> automatically generates the following set of formulas (in SMT-Lib 2.0 syntax):

$$\text{O2F}^{\text{data}^+}(M) \cup \left( \bigcup_{i=1}^n \text{O2F}^{\text{true}}(exp_i) \right) \cup \left( \bigcup_{i=1}^{n+1} \text{O2F}^{\text{def}^+}(exp_i) \right) \cup \text{O2F}^{\text{type}}(exp_{n+1})$$

To validate OCL2FOL<sup>+</sup> we have carried out a number of experiments. In particular, each column in Table 1 shows the result of checking, using Z3, the satisfiability of the set of formulas that results from calling OCL2FOL<sup>+</sup> with **Persons** (*context*), the empty set (*hypothesis*) and the conjunction of some formulas taken from Figure 1 plus the type **true** (*assertion*)

	{1,2}	{1,3}	{2,3}	{4}	{5}	{5,6}	{1,7}	{1,8}	{1,6,8}	{1,9}	{1,6,9}	{1,10}
O2F <sup>true</sup>	unsat	unsat	sat	unsat	sat	unsat	unsat	sat	unsat	sat	unsat	unsat

Table 1. OCL2FOL<sup>+</sup> case study: checking satisfiability

## 5 Conclusions and future work

We have addressed the problem of coping with undefinedness in OCL while at the same time being able to use SMT solvers (e.g., Z3 and Yices) to automatically carry out OCL reasoning. Our solution follows the same principles underlying

OCL2FOL, but consists now of four different mappings which formalize, respectively, when an expression evaluates to `true`, when to `false`, when to `null`, and when to `invalid`. Our solution also differs from [4, 5], since we pursue different goals. In particular, while we are able to support automated OCL reasoning, we can only cover a subset of OCL. On the other hand, the solution presented in [4, 5], although it covers the full OCL language, it can only provide interactive theorem-proving capabilities. As part of this work, we have also implemented our four mappings in a tool called OCL2FOL<sup>+</sup> [2], and we have applied this tool for reasoning about OCL expressions in the presence of undefinedness. For this case study we have used a non-trivial benchmark that we would like to propose as an extension of [10], following their invitation to provide scenarios which are not currently covered: in particular, in their own words, “intensive use of the full semantics of OCL (like the undefined value or collection semantics); this poses a challenge for the lifting to two-valued logics.”

Our future work will follow two main directions. On the one hand, we want to formally prove that our four mappings are correct with respect to the OCL semantics; unfortunately, the OCL semantics for undefinedness has been, in the past, in a constant state of flux. On the other hand, we want to apply OCL2FOL<sup>+</sup> for analyzing ActionGUI models [1], since they contain non-trivial Boolean OCL expressions which should never evaluate to `null` or `invalid`: e.g., *authorization constraints* in permissions or *conditions* in if-then-else statements (which are both expected to always evaluate to either `true` or `false`), and *arguments* in actions (which, when referring to the object upon which an action is to be executed, are expected to never evaluate to `null` or `invalid`).

*Acknowledgements.* We kindly thank Achim Brucker and Marco Guarnieri for our very helpful discussions on the semantics of OCL undefinedness, and we also thank our anonymous reviewers for their insightful comments and suggestions.

## References

1. ActionGUI. The ActionGUI project, 2012. <http://www.actiongui.org>.
2. ActionGUI. The OCL2FOL project, 2012. <http://www.actiongui.org>.
3. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to alloy. *Software and System Modeling*, 9(1):69–86, 2010.
4. A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff. A specification-based test case generation method for UML/OCL. In J. Dingel and A. Solberg, editors, *MoDELS Workshops*, volume 6627 of *LNCS*, pages 334–348, 2010.
5. A. D. Brucker and B. Wolff. Featherweight OCL: a study for the consistent semantics of OCL 2.3 in HOL. In M. Balaban, J. Cabot, M. Gogolla, and C. Wilke, editors, *OCL and Textual Modelling*, pages 19–24. ACM, 2012.
6. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.
7. M. Clavel, M. Egea, and M. A. García de Dios. Checking unsatisfiability for OCL constraints. *ECEASST*, 24, 2009.

8. L. Mendonça de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
9. B. Dutertre and L. Moura. Yices: An SMT solver. <http://yices.csl.sri.com/>, 2008.
10. M. Gogolla, F. Büttner, and J. Cabot. Initiating a Benchmark for UML and OCL Analysis Tools. In Margus Veanes and Luca Viganò, editors, *TAP*, volume 7942 of *LNCS*, pages 115–132. Springer, 2013.
11. M. Kuhlmann and M. Gogolla. From UML and OCL to relational logic and back. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *MoDELS*, volume 7590 of *LNCS*, pages 415–431. Springer, 2012.
12. Object Management Group. *Object Constraint Language specification Version 2.3.1*, January 2012. <http://www.omg.org/spec/OCL/2.3.1>.
13. A. Queralt, A. Artale, D. Calvanese, and E. Teniente. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data & Knowledge Engineering*, 73:1–22, 2012.