OCL2FOL$^+$: Coping with Undefinedness

Carolina Dania and Manuel Clavel

IMDEA Software Institute, Madrid, Spain
Universidad Complutense de Madrid, Madrid, Spain
[carolina.dania,manuel.clavel]@imdea.org

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 [11] 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 [12, 5] 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,[1] 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 [3, 4, 2, 10] though [2, 10] only deals with null.

In [6] 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 [7] or Yices [8]) 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.[2] 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 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$^+$ which, although grounded on the same principles underlying OCL2FOL, is designed to overcome the limitations of the latter with

[1] Intuitively, null represents unknown/undefined values and invalid represents error/exceptions.

[2] Class diagrams and object diagrams are prototypical examples, respectively, of data models and of instances of data models.

regard to undefinedness in OCL.[3] 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$^+$ 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$^+$ for coping with undefinedness in OCL. Next, in Section 4, we report on the status of OCL2FOL$^+$, and on some experiments that we have carried out with this tool. Finally, we draw conclusions in Section 5.

2 OCL2FOL

In [6] we define a mapping, called OCL2FOL, from a subset of OCL[4] into first-order logic. 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 *exp* be a Boolean OCL expression (within the O2F*'s range), whose context is also M. Then,

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

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

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

Next, we briefly discuss these three auxiliary mappings, and then the key mapping O2F*. But before, consider the following remark, whose correctness is based on the correctness of (1) and on the assumption that the evaluation of *exp* 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 *exp* be a Boolean OCL expression (within the O2F*'s range), whose context is also M. Then,

$$\text{Eval}(exp,I) = \mathsf{false} \impliedby \text{O2F}^{\text{env}}(M, I, exp) \not\models \text{O2F}^*(exp)$$

[3] 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$^+$.

[4] This subset includes arbitrary, possibly nested forAll, exists, select, reject, and collect iterator expressions. However, collect-expressions must be followed by asSet. See [6, 1] for further details.

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 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 O2F$^{\text{data}}$(M) includes the formulas:

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

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 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 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 through o's association-end *assoc*, then the set O2F$^{\text{inst}}$(M) includes the following formula:

- $assoc(o,o')$

O2F$^{\text{def}}$(*exp*): Mapping definitions in expressions *exp*

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 *exp* which defines a new collection, the set O2F$^{\text{def}}$(*exp*) 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 *exp* that refers, without naming it, to an object using a property, then set O2F$^{\text{def}}$(*exp*) includes the formula that formalizes that the aforementioned property holds for the element referred to by the new constant.

O2F*(*exp*). 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, collect (followed by asSet) is translated by a predicate formula, whose predicate's definition is generated using O2F$^{\text{def}}$. 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$^{\text{def}}$.

The recursive definition of O2F* is rather technical, and we shall refer to [6, 1] 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 association-ends are friendsOf (x is a friend of y) and friendTo (y is considered a friend by x).

Let *exp* be Person.allInstances()→notEmpty(). Then, O2F*(*exp*) is the formula:

$$\exists(x)(Person(x))$$

where *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 *exp* be Person.allInstances()→exists(p|p.age ≤ 18). Then, O2F*(*exp*) is the following formula:

$$\exists(x)(Person(x) \wedge age(x) \leq 18)$$

where *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 *exp* be Person.allInstances()→select(p|p.age > 18) →isEmpty(). Then, O2F*(*exp*) is the following formula:

$$\forall(x)(\neg Select\#1(x))$$

where x is a logical variable introduced as part of the mapping of the operator isEmpty and *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, using the mapping O2F$^{\text{def}}$, by the following formula:

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

Finally, let *exp* be Person.allInstances()→collect(p|p.friendsOf)→asSet()→notEmpty(). Then, O2F*(*exp*) is the following formula:

$\exists(x)(Collect\#1(x))$

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

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

where *friendsOf* is the predicate that represents the association-end friendsOf.

3 OCL2FOL$^+$: 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 mappings, namely, $O2F^{true}$, $O2F^{false}$, $O2F^{null}$, and $O2F^{inval}$, 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 *exp* 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,

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

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

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

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

$O2F^{data+}(M)$: 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 $O2F^{data+}(M)$ the following formula:

- $IsNull(null) \wedge \neg IsInvalid(null) \wedge \neg IsNull(invalid) \wedge IsInvalid(invalid)$

Objects are neither null nor invalid.[5] Therefore, for every class C in M, the set $O2F^{data+}(M)$ contains the following additional formula:

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

$O2F^{inst+}(I)$: 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 $O2F^{inst+}(I)$ 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 $O2F^{inst+}(I)$ includes the formula:

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

$O2F^{def+}(exp)$: Mapping definitions in expressions exp

Literals of primitive types are also neither null nor invalid. Thus, for every literal l in the given expression exp, the set $O2F^{def+}(exp)$ 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 $O2F^{true}$, $O2F^{false}$, $O2F^{null}$, $O2F^{inval}$. 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 *exp* be Person.allInstances()→notEmpty().

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

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

$O2F^{null}(exp) = \bot$

$O2F^{inval}(exp) = \exists(x)(Person(x) \wedge IsInvalid(x))$

[5] 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.

Notice that, in the logical context provided by $\text{O2F}^{\text{env}+}$, some of the above expressions can be simplified. For example, $\forall(x)(\mathit{Person}(x) \Rightarrow \neg \mathit{IsInvalid}(x))$ can be reduced to $\top$, since objects are neither null nor invalid; for the same reason, $\exists(x)(\mathit{Person}(x) \wedge \mathit{IsInvalid}(x))$ can be reduced to $\bot$. 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 *exp* be Person.allInstances()→ exists(p|p.age ≤ 18).

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

where, if *exp* is p.age <= 18, then

$$\begin{aligned}
&\text{O2F}^{\text{true}}(\mathit{exp}) = \\
&\quad (\text{O2F}^{\text{null}}(\text{p.age}) \wedge \text{O2F}^{\text{null}}(18)) \vee \\
&\quad (\mathit{age}(p) \leq 18 \\
&\qquad \wedge \neg(\text{O2F}^{\text{null}}(\text{p.age}) \vee \text{O2F}^{\text{null}}(18) \vee \text{O2F}^{\text{inval}}(\text{p.age}) \vee \text{O2F}^{\text{inval}}(18))) \\
&\text{O2F}^{\text{false}}(\mathit{exp}) = \\
&\quad \neg(\mathit{age}(p) \leq 18) \\
&\qquad \wedge \neg(\text{O2F}^{\text{null}}(\text{p.age}) \vee \text{O2F}^{\text{null}}(18) \vee \text{O2F}^{\text{inval}}(\text{p.age}) \vee \text{O2F}^{\text{inval}}(18)) \\
&\text{O2F}^{\text{null}}(\mathit{exp}) = \bot \\
&\text{O2F}^{\text{inval}}(\mathit{exp}) = \\
&\quad \text{O2F}^{\text{inval}}(\text{p.age}) \vee \text{O2F}^{\text{inval}}(18) \vee (\text{O2F}^{\text{null}}(\text{p.age}) \wedge \neg(\text{O2F}^{\text{null}}(18)) \\
&\quad \vee (\text{O2F}^{\text{null}}(18) \wedge \neg(\text{O2F}^{\text{null}}(\text{p.age}))
\end{aligned}$$

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 *exp* be Person.allInstances()→select(p|p.age > 18)→isEmpty().

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

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

$O2F^{null}(exp) = \bot$

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

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:

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

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 *exp* be Person.allInstances()→any(p|p.age > 16).oclIsUndefined().

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

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

$O2F^{null}(exp) = \bot$

$O2F^{inval}(exp) = \bot$

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:

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

$$IsNull(Any\#1) \Leftrightarrow (\forall(x)(Person(x) \Rightarrow (O2F^{false}(\text{x.age} > 16) \vee O2F^{null}(\text{x.age} > 16) \vee O2F^{inval}(\text{x.age} > 16))) \wedge \forall(x)(Person(x) \Rightarrow \neg IsInvalid(x)))$$

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

4 Tool support and case study

We have implemented our mappings in a tool, called OCL2FOL$^+$ [1]. OCL2FOL$^+$ takes as input a data model M (the *context*), a set of Boolean OCL expressions $exp_1, \ldots, exp_n$ (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, the

1. Person.allInstances()−>forAll(p|p.age>18)
2. Person.allInstances()−>exists(p|p.age<=18)
3. Person.allInstances()−>exists(p|p.age.oclIsUndefined())
4. Person.allInstances()−>exists(p|p.oclIsUndefined())
5. Person.allInstances()−>forAll(p|p.oclIsUndefined())
6. Person.allInstances()−>notEmpty()
7. Person.allInstances()−>collect(p|p.age)−>asSet()−>exists(a|a.oclIsUndefined())
8. Person.allInstances()−>any(p|p.age>16).oclIsUndefined()
9. Person.allInstances()−>any(p|p.age>16).age.oclIsInvalid()
10. not(Person.allInstances()−>any(p|p.age<16).age.oclIsInvalid())

Fig. 1. OCL2FOL$^+$ case study: sample formulas

	{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$^{\text{true}}$	unsat	unsat	sat	unsat	sat	unsat	unsat	sat	unsat	sat	unsat	unsat

Table 1. OCL2FOL$^+$ case study: checking satisfiability

tool OCL2FOL$^+$ automatically generates the following set of formulas (in SMT-Lib 2.0 syntax):

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

To validate OCL2FOL$^+$ 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$^+$ with **Persons** (*context*), the empty set (*hypothesis*) and the conjunction of some formulas taken from Figure 1 plus the type `true` (*assertion*).

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 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 [3, 4], 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 [3, 4], although it scovers 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$^+$ [1] , 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 [9], 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$^+$ 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. This work is partially supported by the EU FP7-ICT Project "NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems" (256980) by the Spanish Ministry of Science and Innovation Project "DESAFIOS-10" (TIN2009-14599-C03-01), and by Comunidad de Madrid Program "PROMETIDOS-CM" (S2009TIC-1465).

References

1. ActionGUI. ActionGUI and OCL2FOL projects, 2012. http://www.actiongui.org.
2. 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.
3. 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.
4. 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.
5. 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.
6. M. Clavel, M. Egea, and M. A. García de Dios. Checking unsatisfiability for OCL constraints. *ECEASST*, 24, 2009.
7. L. Mendonça de Moura and N. Bjørner. Z3: An efficient SMT solver. In C.R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
8. B. Dutertre and L. Moura. Yices: An SMT solver. http://yices.csl.sri.com, 2008.
9. M. Gogolla, F. Büttner, and J. Cabot. Initiating a Benchmark for UML and OCL Analysis Tools. In M. Veanes and L. Viganò, editors, *TAP*, volume 7942 of *LNCS*, pages 115–132. Springer, 2013.
10. 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.
11. Object Management Group. *Object Constraint Language specification Version 2.3.1*, January 2012. http://www.omg.org/spec/OCL/2.3.1.
12. 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.