OCL2MSFOL: A Mapping to Many-Sorted First-Order Logic for Efficiently Checking the Satisfiability of OCL Constraints

Carolina Dania
IMDEA Software Institute
Madrid, Spain
carolina.dania@imdea.org

Manuel Clavel
IMDEA Software Institute
Madrid, Spain
manuel.clavel@imdea.org

ABSTRACT

In this paper we propose a mapping from OCL to many-sorted first-order logic, called OCL2MSFOL. This new mapping significantly improves our previous results in two key aspects. First, it accepts as input a larger subset of the UML/OCL language; in particular, it supports UML generalization, along with generalization-related OCL operators. Secondly, it generates as output a class of satisfiability problems that can be efficiently handled by SMT solvers with finite model finding capabilities. We report on a non-trivial case study and draw comparisons with related tools.

Keywords

OCL, SMT solvers, finite model finding, satisfiability, automated reasoning

1. INTRODUCTION

Model-Driven Architecture (MDA) supports the development of complex software systems by generating software from models. Logically, the quality of the generated software depends on the quality of the source models. Experience shows that even when using powerful, high-level modelling languages, it is easy to make logical errors and omissions. It is critical then, not only that the modelling language has a well-defined semantics, but also that there is tool support for analyzing the modelled systems' properties.

The Object Constraint Language (OCL) is a strongly-typed, declarative formal language defined by the UML standard to provide the level of conciseness and expressiveness that is required for certain aspects of a design. Expressions in OCL either have a primitive type (integer, real, string, or Boolean), a class type, a tuple type, or a collection type (set, bag, ordered set, or sequence). The language provides standard operators on collections as well as iterators. It also provides the usual operators on primitive types and tuples, and a dot-operator to access object properties, namely, attributes and association-ends. Moreover, OCL provides two constants, null and invalid, to represent *undefinedness*. Intuitively, null represents an unknown or undefined value, whereas invalid represents an error or exception.

With the goal of providing support for UML/OCL reasoning, different mappings to other formalisms have been proposed in the past (see [17] for a systematic review). In each case, the chosen target formalism imposes a different trade-off between expressiveness, termination, automation, and completeness. In particular, most proposals have disregarded OCL undefinedness in order to more easily map OCL to a two-valued formalism; exceptions to this trend are [7] and [4, 15], although the latter only deals with null-related undefinedness.

In [14] a benchmark was introduced for assessing the different UML/OCL reasoning tools. According to this benchmark, tools should be able to address, with respect to a given UML/OCL model, questions regarding *consistency* (whether there exist object diagrams at all), *independence* (whether there are redundant OCL invariants), *consequences* (whether new properties can be derived from stated ones), and *reachability* (whether there exist object diagrams with stated properties). Very naturally, these questions can be formulated as UML/OCL satisfiability problems.

In previous works we proposed two mappings from OCL to first-order logic (FOL) that supported the use of Satisfiability Modulo Theories (SMT) solvers for checking UML/OCL satisfiability problems.[1] In [10] we proposed a first mapping from UML/OCL to first-order logic, called OCL2FOL, which did not support UML generalization or OCL undefinedness. In [11] we proposed a second mapping, called OCL2FOL$^+$, which did take into account OCL undefinedness, but did not support UML generalization. Moreover, OCL2FOL$^+$ turned out to be rather inefficient in practice, since SMT solvers would often return *unknown*, as a consequence of two facts: first, that non-trivial OCL constraints contain expressions that are naturally mapped to quantified formulas (since they refer to *all* the objects in a class, for example), and, secondly, that techniques for dealing with quantified formulas in SMT are generally incomplete.

To overcome this limitation, we decided to use SMT solvers along with finite model finding methods for checking the satisfiability of the formulas resulting from our mapping. In particular, we opted for using the SMT solver CVC4 [6], which has a finite model finding method [24] fully integrated

MODELS '16, October 02-07, 2016, Saint-Malo, France
© 2016 ACM. ISBN 978-1-4503-4321-3/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2976767.2976774

[1]SMT solvers generalize Boolean satisfiability (SAT) by incorporating equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other first-order theories.

with its SMT solver architecture. The finite model finding method implemented in CVC4 requires, however, that quantified variables in the input problem always range over finite domains. OCL2FOL$^+$ could not satisfy this requirement, since its target formalism was *unsorted* FOL: variables in quantified formulas generated by OCL2FOL$^+$ range over a single, infinite domain that includes the integer numbers. By switching to *many-sorted* FOL (MSFOL), we were able to satisfy the aforementioned requirement: variables in quantified formulas range now over the domain of a distinguished sort, called *Classifier*, which essentially contains the objects in an object diagram and the undefined values (but not the integer numbers or the strings), and which, for the purpose of UML/OCL verification, can be considered as finite (object diagrams can be assumed to contain only a finite number of objects). Finally, many-sorted FOL provides a more adequate target formalism than unsorted FOL for mapping UML generalization and generalization-related OCL operations.

In summary, we propose a new mapping from OCL to many-sorted first-order logic, called OCL2MSFOL, which successfully overcomes the limitations of our previous mappings. First, it accepts as input a significantly larger subset of the UML/OCL language; in particular, it supports UML generalization, along with the generalization-related OCL operators. Secondly, it generates as output a class of satisfiability problems that are amenable to checking by using SMT solvers with finite model finding capabilities.

Organization.
In Section 2 we introduce our mapping from UML class diagrams to MSFOL theories and, in Section 3 we explain our mapping from OCL constraints to MSFOL formulas. In Section 4 we discuss how to check the satisfiability of OCL constraints using SMT solvers, based on our mapping from UML/OCL to MSFOL. Also, we present a tool that supports our methodology and we provide a preliminary benchmark. In Section 5 we report on a non-trivial case study, involving a UML class diagram with 9 classes, 3 generalizations, 24 attributes, 10 associations, and 38 invariants. We conclude with a discussion of related work and draw conclusions.

2. FROM DATA MODELS TO MSFOL

OCL is a contextual language. We call *data models* the contexts supported by our mapping OCL2MSFOL. Data models are a subclass of UML class diagrams. In particular, data models support *generalizations*, binary *associations*, and *attributes* either of primitive types Integer and String or of class types. In Definition 1 (Appendix A) we formally define our notion of data models.

Our mapping from OCL to MSFOL builds upon a base mapping from data models to MSFOL theories, called $\mathrm{o2f}_{\mathrm{data}}$. Let $\mathcal{D}$ be a data model. In a nutshell, $\mathrm{o2f}_{\mathrm{data}}(\mathcal{D})$ contains:

- The sorts *Int* and *String*, whose intended meaning is to represent the integer numbers and the strings.
- The constants *nullInt*, *nullString*, *invalInt*, and *invalString*, whose intended meaning is to represent null and invalid for integers and strings.
- The sort *Classifier*, whose intended meaning is to represent *all* the objects in an instance of $\mathcal{D}$, as well as null and invalid for objects.

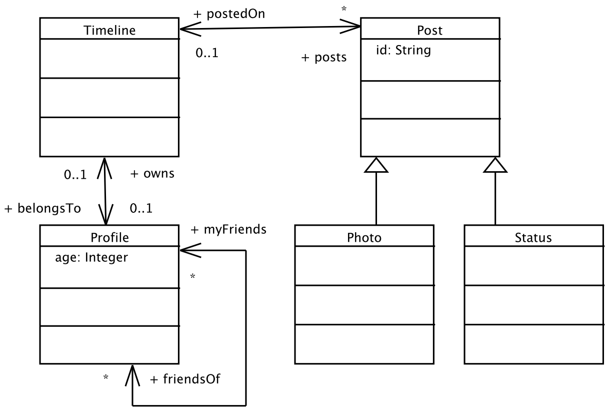

Figure 1: A data model for a basic social network.

- The sort *Type*, whose intended meaning is to represent the type identifiers declared in $\mathcal{D}$.
- For each class c in $\mathcal{D}$, a unary predicate c, whose intended meaning is to define the objects of the class c in an instance of $\mathcal{D}$
- For each attribute at in $\mathcal{D}$, a function at, whose intended meaning is to define the values of the attribute at in the objects in an instance of $\mathcal{D}$.
- For each binary association aso in $\mathcal{D}$ with association-ends as (with multiplicity *) and as' (with multiplicity *), a binary predicate as_as', whose intended meaning is to define the links through the association aso between objects in an instance of $\mathcal{D}$.[2]
- The axioms that constrain the meaning of the aforementioned sorts, constants, predicates, and functions.

In the following example we illustrate the mapping $\mathrm{o2f}_{\mathrm{data}}$. A partial definition of $\mathrm{o2f}_{\mathrm{data}}$, which basically contains the clauses required for the example below, is given in Definition 2 (Appendix A). The interested reader can find the full definition of the mapping $\mathrm{o2f}_{\mathrm{data}}$ in [12].

Example 1. Consider the data model SSN shown in Figure 1, which models a basic social network. Then, the MSFOL theory $\mathrm{o2f}_{\mathrm{data}}(\mathsf{SSN})$ contains, among other elements:

- The constants *nullClassifier* and *invalClassifier* of sort Classifier, along with the axiom:

  $$\neg(\mathit{nullClassifier} = \mathit{invalClassifier}).$$

- The constants *nullInt* and *invalInt* of sort *Int*, along with the axiom:

  $$\neg(\mathit{nullInt} = \mathit{invalInt}).$$

[2]For associations with both association-ends with multiplicities 0..1, our mapping declares a function for each association-end, instead of a predicate for the association. Then, for associations with one association-end with multiplicity * and the other with multiplicity 0..1, our mapping declares a binary predicate for the association-end with multiplicity * and a function for the one with multiplicity 0..1.

- The predicate $\mathsf{Profile}\colon \mathit{Classifier} \to \mathit{Bool}$, along with the axioms:

  $\forall(x)(\mathit{Profile}(x) \implies$
  $\quad\neg(\mathit{Photo}(x) \vee \mathit{Status}(x) \vee \mathit{Timeline}(x) \vee \mathit{Post}(x)))$.
  $\neg(\mathit{Profile}(\mathit{nullClassifier}) \vee \mathit{Profile}(\mathit{invalClassifier}))$.

- The function $\mathit{age} : \mathit{Classifier} \to \mathit{Int}$, along with the axioms:

  $\mathit{age}(\mathit{nullClassifier}) = \mathit{invalInt}$.
  $\mathit{age}(\mathit{invalClassifier}) = \mathit{invalInt}$.
  $\forall(x)(\mathit{Profile}(x) \implies \neg(\mathit{age}(x) = \mathit{invalInt}))$.

- The predicate $\mathit{myFriends\_friendsOf}\colon \mathit{Classifier} \times \mathit{Classifier} \to \mathit{Bool}$, along with the axioms:

  $\forall(x, y)(\mathit{myFriends\_friendsOf}(x, y)$
  $\quad\Leftrightarrow (\mathit{Profile}(x) \wedge \mathit{Profile}(y)))$.

- The constants $\mathit{Post}_{\mathrm{type}}$, $\mathit{Photo}_{\mathrm{type}}$, and $\mathit{Status}_{\mathrm{type}}$ of sort Type, along with the axioms:

  $\neg(\mathit{Post}_{\mathrm{type}} = \mathit{Photo}_{\mathrm{type}})$.
  $\neg(\mathit{Post}_{\mathrm{type}} = \mathit{Status}_{\mathrm{type}})$.
  $\neg(\mathit{Photo}_{\mathrm{type}} = \mathit{Status}_{\mathrm{type}})$.

- The predicate $\mathit{oclIsKindOf}\colon \mathit{Classifier} \times \mathit{Type} \to \mathit{Bool}$, along with the axioms:

  $\forall(x)(\mathit{oclIsKindOf}(x, \mathit{Post}_{\mathrm{type}})$
  $\quad\Leftrightarrow (\mathit{Post}(x) \vee \mathit{Photo}(x) \vee \mathit{Status}(x)))$.
  $\forall(x)(\mathit{oclIsKindOf}(x, \mathit{Photo}_{\mathrm{type}}) \Leftrightarrow \mathit{Photo}(x))$.
  $\forall(x)(\mathit{oclIsKindOf}(x, \mathit{Status}_{\mathrm{type}}) \Leftrightarrow \mathit{Status}(x))$.

3. FROM OCL TO MSFOL

OCL2MSFOL is designed for checking the satisfiability of OCL constraints: it accepts as input OCL *Boolean* expressions, and only deals with non-Boolean expressions inasmuch as they appear as subexpressions of Boolean expressions.

The mappings $\mathrm{o2f}_{\mathrm{true}}$, $\mathrm{o2f}_{\mathrm{false}}$, $\mathrm{o2f}_{\mathrm{null}}$, *and* $\mathrm{o2f}_{\mathrm{inval}}$.

In the presence of undefinedness, OCL Boolean expressions can evaluate not only to true and false but also to null or invalid. To cope with four Boolean values in a two-valued logic like MSFOL, we define four mappings, namely, $\mathrm{o2f}_{\mathrm{true}}$, $\mathrm{o2f}_{\mathrm{false}}$, $\mathrm{o2f}_{\mathrm{null}}$, and $\mathrm{o2f}_{\mathrm{inval}}$, which formalize when a Boolean expression evaluates to true, when to false, when to null, and when to invalid. We define these mappings by *structural recursion*. In the recursive case, when the subexpression is a non-Boolean type, we call an auxiliary mapping, $\mathrm{o2f}_{\mathrm{eval}}$, which we will discuss below. For now, it is sufficient to know that $\mathrm{o2f}_{\mathrm{eval}}$ returns a term when its argument is an expression of a class type or of type Integer or String, and that it returns a predicate when its argument is an expression of a set type.[3]

In the following examples we illustrate the recursive definitions of the mappings $\mathrm{o2f}_{\mathrm{true}}$, $\mathrm{o2f}_{\mathrm{false}}$, $\mathrm{o2f}_{\mathrm{null}}$, and $\mathrm{o2f}_{\mathrm{inval}}$. A partial definition of these mappings, which basically contains the clauses required for the examples below, is given in Appendix B. The interested reader can find the full definition of these mappings in [12].

[3]We assume that all non-Boolean subexpressions have either a class type, a primitive type (either Integer or String), or a set type.

Example 2. Consider the Boolean expression:

$\mathsf{Profile.allInstances()}{\to}\mathsf{notEmpty()}$.

Then,

$\mathrm{o2f}_{\mathrm{true}}(\mathsf{Profile.allInstances()}{\to}\mathsf{notEmpty()})$
$\quad= \exists(x\colon \mathit{Classifier})(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()})(x))$
$\qquad\wedge \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathsf{Profile.allInstances()}))$
$\quad= \exists(x\colon \mathit{Classifier})(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()})(x))$
$\qquad\wedge \neg(\bot)$.

Example 3. Consider the Boolean expression:

$\mathsf{Profile.allInstances()}{\to}\mathsf{forAll(p|not(p.age.oclIsUndefined()))}$.

Then,

$\mathrm{o2f}_{\mathrm{false}}(\mathsf{Profile.allInstances()}$
$\qquad\qquad{\to}\mathsf{forAll(p|not(p.age.oclIsUndefined())))}$
$\quad= \exists(p\colon \mathit{Classifier})(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()})(p)$
$\qquad\qquad\wedge\ \mathrm{o2f}_{\mathrm{false}}(\mathsf{not(p.age.oclIsUndefined())})$
$\qquad\wedge \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathsf{Profile.allInstances()})))$,

where

$\mathrm{o2f}_{\mathrm{false}}(\mathsf{not(p.age.oclIsUndefined())})$
$\quad= \mathrm{o2f}_{\mathrm{true}}(\mathsf{p.age.oclIsUndefined()})$
$\quad= \mathrm{o2f}_{\mathrm{null}}(\mathsf{p.age}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathsf{p.age})$
$\quad= \mathrm{o2f}_{\mathrm{eval}}(\mathsf{p.age}) = \mathit{nullClassifier}$
$\qquad\vee\ (\mathrm{o2f}_{\mathrm{null}}(\mathsf{p}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathsf{p}))$
$\quad= \mathrm{o2f}_{\mathrm{eval}}(\mathsf{p.age}) = \mathit{nullClassifier}$
$\qquad\vee(p = \mathit{nullClassifier} \vee p = \mathit{invalClassifier})$.

Example 4. Consider the Boolean expression:

$\mathsf{Profile.allInstances()}{\to}\mathsf{select(p|p.age.oclIsUndefined())}$
$\quad{\to}\mathsf{notEmpty()}$.

Then,

$\mathrm{o2f}_{\mathrm{true}}(\mathsf{Profile.allInstances()}$
$\qquad\qquad{\to}\mathsf{select(p|p.age.oclIsUndefined())}{\to}\mathsf{notEmpty()})$
$\quad= \exists(x\colon \mathit{Classifier})(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()}$
$\qquad\qquad{\to}\mathsf{select(p|p.age.oclIsUndefined())})(x))$
$\qquad\wedge \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathsf{Profile.allInstances()}$
$\qquad\qquad{\to}\mathsf{select(p|p.age.oclIsUndefined())}))$
$\quad= \exists(x\colon \mathit{Classifier})(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()}$
$\qquad\qquad{\to}\mathsf{select(p|p.age.oclIsUndefined())})(x))$
$\qquad\wedge \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathsf{Profile.allInstances()}))$
$\quad= \exists(x\colon \mathit{Classifier})(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()}$
$\qquad\qquad{\to}\mathsf{select(p|p.age.oclIsUndefined())})(x))$
$\qquad\wedge \neg(\bot)$.

The mapping $\mathrm{o2f}_{\mathrm{eval}}$.

In the definition of the mapping $\mathrm{o2f}_{\mathrm{eval}}$ we distinguish three classes of non-Boolean expressions. We explain below the differences between these classes and illustrate the mapping $\mathrm{o2f}_{\mathrm{eval}}$ for the first two classes. A partial definition of $\mathrm{o2f}_{\mathrm{eval}}$, which basically contains the clauses required for the examples below, is given in Appendix C. The interested reader can find the full definition of these mappings in [12].

The first class is formed by variables and by expressions that *denote primitive values and objects.* Expressions denoting primitive values and objects are basically the literals (integers or strings), the arithmetic expressions, the expressions denoting operations on strings, and the dot-expressions for attributes or association-ends with multiplicity 0..1. Variables are mapped to MSFOL variables of the appropriate sort. Expressions denoting primitive values and objects are mapped by $\mathrm{o2f}_{\mathrm{eval}}$ following the definition of the mapping $\mathrm{o2f}_{\mathrm{data}}$. The output of the mapping $\mathrm{o2f}_{\mathrm{eval}}$ for this first class of non-Boolean expressions is always an MSFOL term.

Example 5. Consider the non-Boolean expression:

p.age,

where p is a variable of type Profile. Then,

$$\mathrm{o2f}_{\mathrm{eval}}(\mathsf{p.age}) \\ = \mathit{age}(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{p})) = \mathit{age}(p),$$

where p is a variable of sort *Classifier*.

The second class of non-Boolean expressions is formed by the expressions that *define sets*. These expressions are basically the allInstances-expressions, the select and reject-expressions, the including and excluding-expressions, the intersection and union-expressions, and the collect-expressions. Each expression *expr* in this class is mapped by $\mathrm{o2f}_{\mathrm{eval}}$ to a *new* predicate, denoted as $[\mathit{expr}]$. This predicate formalizes the set defined by the expression *expr* and its definition is generated by calling another mapping, $\mathrm{o2f}_{\mathrm{def\_c}}$, over the expression *expr*.

Example 6. Consider the non-Boolean expression:

Post.allInstances().

Then,

$$\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Post.allInstances()}) \\ = [\mathsf{Post.allInstances()}],$$

where the new predicate

[Post.allInstances()]

is defined by $\mathrm{o2f}_{\mathrm{def\_c}}$ as follows:

$$\forall(x{:}\mathit{Classifier})([\mathsf{Post.allInstances()}] \\ \Leftrightarrow (\mathit{Post}(x) \vee \mathit{Photo}(x) \vee \mathit{Status}(x))).$$

Example 7. Consider the non-Boolean expression:

Profile.allInstances()->select(p|p.age.oclIsUndefined).

Then,

$$\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()} \\ \rightarrow\mathsf{select(p|p.age.oclIsUndefined())}) \\ = [\mathsf{Profile.allInstances()}\rightarrow\mathsf{select(p|p.age.oclIsUndefined())}],$$

where the new predicate

[Profile.allInstances()→select(p|p.age.oclIsUndefined())]

is defined by $\mathrm{o2f}_{\mathrm{def\_c}}$ as follows:

$$\forall(p{:}\mathit{Classifier})([\mathsf{Profile.allInstances()} \\ \rightarrow\mathsf{select(p|p.age.oclIsUndefined())}](p) \\ \Leftrightarrow \\ (\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()})(p) \\ \wedge\ \mathrm{o2f}_{\mathrm{true}}(\mathsf{p.age.oclIsUndefined()}))) \\ = \forall(p{:}\mathit{Classifier})([\mathsf{Profile.allInstances()} \\ \rightarrow\mathsf{select(p|p.age.oclIsUndefined())}](p) \\ \Leftrightarrow \\ [\mathsf{Profile.allInstances()}](p) \\ \wedge\ \mathrm{o2f}_{\mathrm{eval}}(\mathsf{p.age}) = \mathit{nullClassifier} \\ \vee\ (p = \mathit{nullClassifier} \vee p = \mathit{invalClassifier})) \\ = \forall(p{:}\mathit{Classifier})([\mathsf{Profile.allInstances()} \\ \rightarrow\mathsf{select(p|p.age.oclIsUndefined())}](p) \\ \Leftrightarrow \\ [\mathsf{Profile.allInstances()}](p) \\ \wedge\ \mathit{age}(p) = \mathit{nullClassifier} \\ \vee\ (p = \mathit{nullClassifier} \vee p = \mathit{invalClassifier})).$$

where the new predicate [Profile.allInstances()] is defined by $\mathrm{o2f}_{\mathrm{def\_c}}$ as follows:

$$\forall(x{:}\mathit{Classifier})([\mathsf{Profile.allInstances()}] \\ \Leftrightarrow\ \mathit{Profile}(x)).$$

The third class of non-Boolean expressions is formed by the expressions that *distinguish an element from a set*. These expressions are, basically, the any, max, and min-expressions. Each expression *expr* in this class is mapped by $\mathrm{o2f}_{\mathrm{eval}}$ to a *new* function, denoted as $[\mathit{expr}]$, which represents the element referred to by *expr*. To generate the axioms defining $[\mathit{expr}]$, we call another mapping, $\mathrm{o2f}_{\mathrm{def\_o}}$, over *expr*.

In what follows we denote by $\mathrm{o2f}_{\mathrm{def}}(\mathit{expr})$ the set of axioms that result from applying $\mathrm{o2f}_{\mathrm{def\_c}}$ and $\mathrm{o2f}_{\mathrm{def\_o}}$ to the corresponding non-Boolean subexpression in *expr*. Notice that, in particular, for each literal integer i and literal string st in *expr*, $\mathrm{o2f}_{\mathrm{def\_o}}$ generates the following axioms:

$$\mathrm{o2f}_{\mathrm{dfn\_o}}(i) = \{\neg(i = \mathit{nullInt}) \wedge \neg(i = \mathit{invalInt})\}. \\ \mathrm{o2f}_{\mathrm{dfn\_o}}(st) = \{\neg(i = \mathit{nullString}) \wedge \neg(i = \mathit{invalString})\}$$

Current limitations of our mapping.

The key limitation of OCL2MSFOL comes from the fact that expressions defining collections are mapped, as we have explained, to predicates. Although these new predicates are defined so as to capture the property that distinguishes the elements belonging to the given collection, this is not sufficient for reasoning about the *size* of this collection, or about the *multiplicity* or the *ordering* of its elements. Because of this, OCL2MSFOL cannot support, in general, size-expressions or expressions of collection types different from set types. Fortunately, we are not finding this limitation hindering the applicability of our mapping. Other limitations of OCL2MSFOL are mostly due to time constraints, and will be soon corrected, including the current lack of support for attributes of type Boolean and for multiplicities of the form $[n..m]$, where n, m are natural numbers. In the first case, the corresponding terms t of type Boolean would be replaced by formulas of the form $t = \top$. In the second case, the data model would be extended with the corresponding invariants. Finally, notice that it is also fairly trivial to extend our mapping to support n-ary associations.

4. CHECKING SATISFIABILITY

WebOCL2MSFOL [29] is a Java Web Application that implements OCL2MSFOL. More specifically, it takes as input a data model $\mathcal{D}$ and a set of $\mathcal{D}$-constraints $\mathcal{I}$, and returns a file containing the following MSFOL theory:

$$\mathrm{o2f}_{\mathrm{data}}(\mathcal{D}) \cup \left(\bigcup_{inv\in\mathcal{I}} \mathrm{o2f}_{\mathrm{def}}(inv)\right) \quad (1)$$
$$\cup \left(\bigcup_{inv\in\mathcal{I}} \{\mathrm{o2f}_{\mathrm{true}}(inv)\}\right)$$

Additionally, the user can introduce a $\mathcal{D}$-Boolean expression *expr* and request WebOCL2MSFOL to map *expr* to MSFOL using either $\mathrm{o2f}_{\mathrm{true}}$, $\mathrm{o2f}_{\mathrm{false}}$, $\mathrm{o2f}_{\mathrm{null}}$, or $\mathrm{o2f}_{\mathrm{inval}}$. The result will be a file containing the MSFOL theory (1) extended with the following axioms:

$$\mathrm{o2f}_{\mathrm{def}}(expr) \cup \{\mathrm{o2f}_{map}(expr)\}. \quad (2)$$

where *map* is either true, false, null, or inval, depending on the user's choice.

The typical use case for WebOCLMSFOL is as follows. Suppose a data-model $\mathcal{D}$, with invariants $\mathcal{I}$, and a Boolean $\mathcal{D}$-expression *expr*. Then, to check whether there exists a valid instance of $\mathcal{D}$ in which *expr* evaluates to true, we do the following: i) we input in WebOCL2MSFOL the data model $\mathcal{D}$, the set of invariants $\mathcal{I}$, and the expression *expr*; ii) we select the option true; and iii) we input the file generated by WebOCL2MSFOL into our SMT solver of choice. If the SMT solver returns sat, then we know that such an instance of $\mathcal{D}$ exists; if the SMT solver returns unsat then we know that no such an instance of $\mathcal{D}$ exists; and, finally, if the SMT solver returns unknown, then we know that it remains unknown whether such an instance of $\mathcal{D}$ exists. The process is entirely similar if we want to know whether there exists a valid instance of $\mathcal{D}$ in which an expression *expr* evaluates to false, null, or inval; the only difference is that, instead of true, we will select, respectively, false, null, or inval.

To conclude this section we introduce a benchmark for checking the satisfiability of OCL constraints, and report on our results. All checks are ran on a laptop computer, with an Intel Core i7 processor running at 3.1GHz with 8Gb of RAM. As back-end theorem-provers, we use Z3 [13] (version 4.4.1), and CVC4 [6] (version 1.5-prerelease). In the case of Z3, we use its default setting, but in the case of CVC4, we use two different settings, namely, with and without the option finite-model-find. In what follows, we refer to the latter as CVC4 Finite Model (or CVC4fm, for short).

The data model for our benchmark is the basic social network model shown in Figure 1. The Boolean expressions that we consider in our benchmark are the following:

1. Profile.allInstances()→forAll(p|p.age>18)
2. Profile.allInstances()→exists(p|p.age<=18)
3. Profile.allInstances()→exists(p|p.age.oclIsUndefined())
4. Profile.allInstances()→exists(p|p.oclIsUndefined())
5. Profile.allInstances()→forAll(p|p.oclIsUndefined())
6. Profile.allInstances()→notEmpty()
7. Profile.allInstances()→collect(p|p.age)→asSet()
   →exists(a|a.oclIsUndefined())
8. Profile.allInstances()→any(p|p.age>16).oclIsUndefined()
9. Profile.allInstances()→any(p|p.age>16).age.oclIsInvalid()
10. not(Profile.allInstances()
    →any(p|p.age<16).age.oclIsInvalid())
11. Status.allInstances()→notEmpty()
12. Post.allInstances()→forAll(p|
    Photo.allInstances()→exists(q|p.id=q.id))
13. Post.allInstances()→forAll(p|not(p.id.oclIsUndefined()))
14. Status.allInstances()→notEmpty()
15. Status.allInstances()→isEmpty()
16. Photo.allInstances()→notEmpty()
17. Photo.allInstances()→isEmpty()
18. Post.allInstances()→notEmpty()
19. Post.allInstances()→isEmpty()
20. Post.allInstances()
    →forAll(p|Photo.allInstances()->exists(q|p.id=q.id))
21. Photo.allInstances()→forAll(p|p.oclIsKindOf(Post))
22. Photo.allInstances()→forAll(p|p.oclIsKindOf(Timeline))
23. Post.allInstances()→forAll(p|p.oclIsTypeOf(Timeline))
24. Post.allInstances()
    →forAll(p|not(p.oclIsTypeOf(Post)))
25. 2.oclIsUndefined()
26. Post.allInstances()→forAll(p|
    Post.allInstances()→forAll(q|p<>q implies p.id<>q.id)
27. Profile.allInstances()
    →forAll(p|p.myFriends→notEmpty())

In Table 1 we show the result of checking, using WebOCL2MSFOL, the satisfiability of different subsets of our benchmark's Boolean expressions. We have grouped all our checks in two tables: (1a) contains the checks related to undefinedness, while (1b) contains the checks related to UML generalization. In both cases, the first column indicates the set of Boolean expressions to be checked for satisfiability; the second column indicates the expected result, according to our understanding of the semantics of OCL; and the third, fourth, and fifth column indicate, respectively, the time (in milliseconds) taken by CVC4, Z3, and CVC4 Finite Model to return the expected result, or '—', in the case they return unknown. Notice that CVC4 Finite Model is able to return the expected result in all cases, while Z3 and CVC4 return *unknown* in some cases.

5. CASE STUDY

In this section we carry out a case study for checking the efficiency of CVC4 Finite Model when reasoning about UML/OCL using our mapping OCL2MSFOL.[4]

The data model for our case study, shown in Figure 2, models a basic eHealth management system. It contains 9 classes, 3 generalizations, 24 attributes, and 10 associations. It also contains 38 invariants, which can be grouped in the 5 categories:[5]

G1. Invariants stating the non-emptiness of certain classes. For example, *There must be at least one medical center.*

MedicalCenter.allInstances()→notEmpty().

There are 9 invariants in this category, one for each class in the data model: namely, MedicalCenter, Department, Professional, Director, Doctor, Nurse, Referral, Patient, and ContactInfo.

G2. Invariants stating the definedness of certain attributes. For example, *The name of a professional cannot be undefined.*

[4]As discussed before, pure SMT solvers like Z3 or CVC4, i.e., SMT solvers without finite model finding capabilities, typically return *unknown* when checking the satisfiability of nontrivial OCL constraints using our mapping OCL2MSFOL.

[5]Notice that the given set of invariants is not intended to be complete.

		CVC4	Z3	CVC4fm
{1,2}	unsat	161	24	48
{1,3}	unsat	173	13	22
{2,3}	sat	—	16	25
{4}	unsat	138	15	27
{5}	sat	—	17	22
{5,6}	unsat	172	13	30
{1,7}	unsat	237	14	30
{1,8}	sat	—	18	25
{1,6,8}	unsat	198	16	26
{1,9}	sat	—	18	25
{1,6,9}	unsat	200	19	29
{1,10}	unsat	203	18	30
{12}	sat	—	169	27
{11,12,13}	sat	—	24	174

(a) Undefinedness-related (times in ms)

		CVC4	Z3	CVC4fm
{14,20}	sat	—	105	28
{16,20}	sat	—	466	32
{17,20}	sat	—	14	22
{14,17,20}	unsat	239	13	26
{16,19}	unsat	168	16	28
{21}	sat	—	17	27
{22}	sat	—	199	24
{16,22}	unsat	149	18	25
{16,23}	unsat	148	16	26
{15,17,18,24}	unsat	250	15	35
{25}	unsat	63	58	24
{11,12,13,18}	sat	—	—	27
{6,27}	sat	—	—	26
{11,12,13,18,26}	unsat	352	13	25

(b) Generalization-related (times in ms)

Table 1: Checking satisfiability of OCL constraints.

```
Professional.allInstances()
  →forAll(p|not(p.name.oclIsUndefined())).
```

There are 11 invariants in this category, stating the definedness of the attributes name (MedicalCenter), city (MedicalCenter), country (MedicalCenter), director (MedicalCenter), name (Department), name (Professional), surname (Professional), login (Professional), password (Professional), contactInfo (Patient), and license (Nurse),

G3. Invariants stating the uniqueness of certain data with respect to certain attributes. For example, *There cannot be two different doctors with the same medical license number.*

```
Doctor.allInstances()→forAll(d1|
  Doctor.allInstances()→forAll(d2|
    not(d1=d2) implies not(d1.license=d2.license)).
```

There are 5 invariants in this category, stating the uniqueness of certain data with respect to different attributes. In particular, data of the class MedicalCenter, when considering together address, zipCode, city, and country; data of the class Professional, with respect to login; data of the class Doctor, with respect to license; data of the class Nurse, with respect to license; and data of the class Referral, when considering together patient, referringTo, and referredBy.

G4. Invariants stating the non-emptiness of certain association-ends. For example, *Every medical center should have at least one employee.*

```
MedicalCenter.allInstances()
  →forAll(m|m.employees→notEmpty()).
```

There are in total 6 invariants in this category, stating the non-emptiness of the association-ends employees (MedicalCenter), belongsTo (Department), doctors (Department), nurses (Department), patient (Referral), referredBy (Referral), doctor (Patient), and department (Patient).

G5. Other invariants: namely,

- *A patient should be treated in a department where his/her doctor works.*

```
Patient.allInstances()→forAll(p|
  p.doctor.departments→exists(d|d=p.department))
```

- *A professional cannot have an empty string as password.*

```
Professional.allInstances()→forAll(p|
  not(p.password = ''))
```

- *A professional cannot have an empty string as login.*

```
Professional.allInstances()→forAll(p|
  not(p.login = ''))
```

- *If a doctor's status is 'unavailable', then he/she should have a substitute different from him/herself.*

```
Doctor.allInstances()→forAll(d|
  d.status='unavailable' implies
    (not(d.substitutedBy.oclIsUndefined() or
      d.substitutedBy = d)))
```

- *If doctor's status is 'available', then he/she should not have any substitute.*

```
Doctor.allInstances()→forAll(d|
  d.status='available' implies
    d.substitutedBy.oclIsUndefined())
```

- *If a doctor is a substitute of other doctors, his/her status should be 'available'.*

```
Doctor.allInstances()→forAll(d|
  d.substitutions→notEmpty() implies
    d.status="available")
```

- *If a referral indicates both the patient and the doctor whom the patient is referred to, then the doctor who is referring the patient cannot be the same than the doctor whom the patient is referred to.*

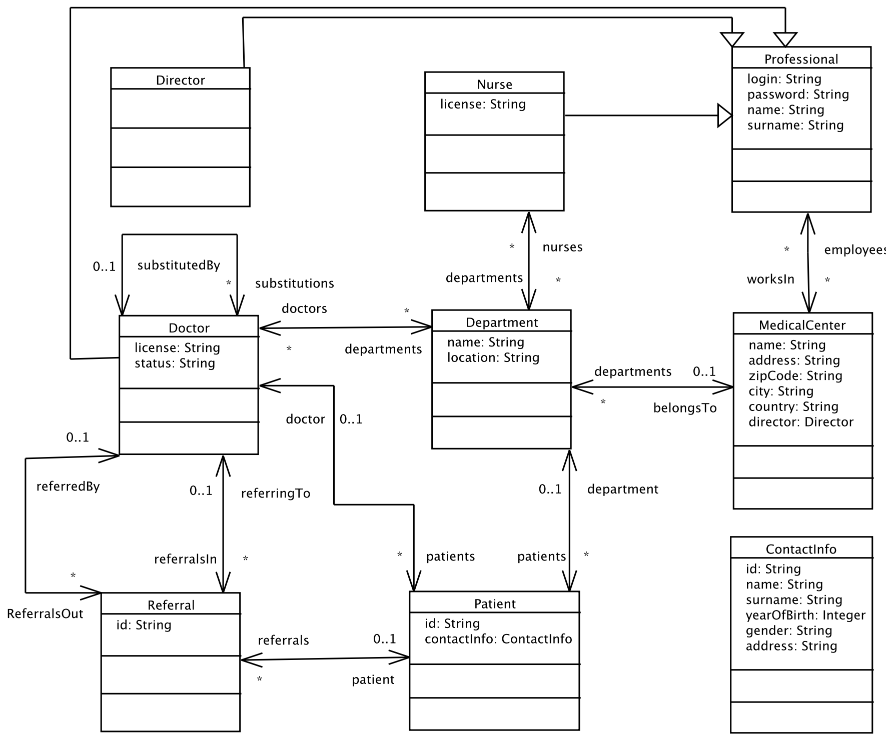

Figure 2: EHR: a data model for a basic eHealth Record Management System.

Referral.allInstances()→forAll(r|
  not(r.patient.oclIsUndefined() and
      r.referringTo.oclIsUndefined()) implies
    not(r.referringTo = r.referredBy))

In what follows we report on the results obtained in our case study. The first experiment we carried out was to check whether there exists an instance of the case study's data model satisfying all the given invariants. Using CVC4 Finite Model we obtained the answer in 6 seconds. In particular, we show in Figure 3 the valid instance of the case study's data-model automatically generated by CVC4 Finite Model.

The second experiment consisted in checking whether there exists a valid instance of the case study's data model for which the following also holds:

- *There exists only one doctor and his/her status is "unavailable".*

  Doctor.allInstances()→exists(d1|
    d1.status='unavailable' and
      Doctor.allInstances()→forAll(d1|d1=d2)).

Using CVC4 Finite Model we obtained unsat in about 4 seconds. This is the expected result, since there is an invariant stating that "if a doctor's status is 'unavailable', then he/she should have a substitute different from him/herself."

Finally, the third experiment consisted in checking whether there exists a valid instance of the case study's data model for which the following holds:

- *There exists only one doctor*

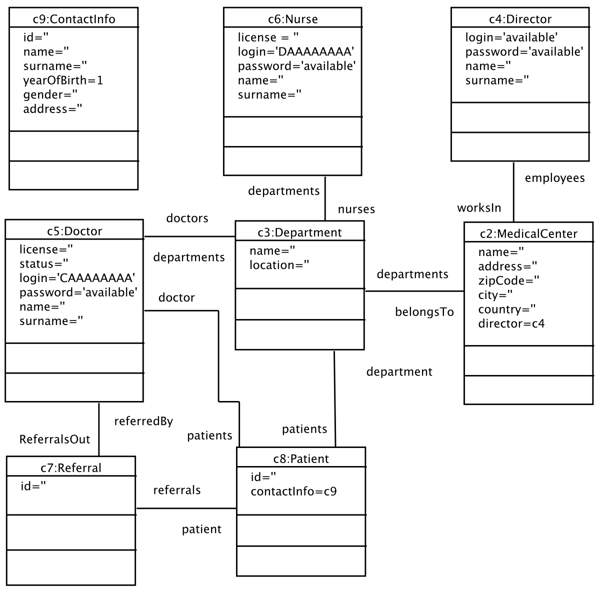

Figure 3: Automatically generated instance of the case study's data model satisfying all the invariants.

	Mapping	Target formalism
G1	FiniteSAT [19]	System of Linear Inequalities
	DL [5]	Description Logics, CSP
	MathForm [28]	Mathematical Notation
G2	UMLtoCSP [9]	CSP
	EMFtoCSP [16]	CSP
	AuRUS [22]	FOL
	OCL2FOL [10]	FOL
	OCL-Lite [21]	Description Logics
	BV-SAT [27]	Relation Logic
	PVS [23]	HOL
	CDOCL-HOL [2]	HOL
	KeY [1]	Dynamic Logic
	Object-Z [25]	Object-Z
	UML-B [20]	B
G3	UML2Alloy [3]	Relation Logic
	USE [15]	Relation Logic
G4	HOL-OCL [8]	HOL
	OCL2FOL$^+$ [11]	FOL

Table 2: Other mappings from UML/OCL to other formalism.

```
Doctor.allInstances→exists(d1|
  Doctor.allInstances→forAll(d2|d1=d2)).
```

Using CVC4 Finite Model we obtained sat in 7 seconds. This may be an unexpected result, since there is an invariant stating that "that there must be at least one referral" and there is another invariant stating that "if a referral has defined both the patient and the doctor to whom the patient is referred to, then the doctor referring the patient cannot be the same than the doctor the patient is referred to". However, the result returned by CVC4 Finite Model is correct, since there is no invariant stating that every referral has defined the doctor to whom the patient is referred to. Therefore, it is certainly possibly a valid instance of the case study's data model basically containing only one doctor and only one referral in which the doctor to whom the patient is referred to is not defined. This is in fact the instance automatically generated by CVC4 Finite Model when returning sat in this experiment. Of course, if we want to "correct" this, we can simply add to the case study's data model the following invariant:

```
Referral.allInstances()→forAll(r|
  not(r.referringTo.oclIsUndefined()))
```

Then, if we run again the experiment, the answer returned by CVC4 Finite Model, in about 4 seconds, is unsat.

6. RELATED WORK

In Table 2 we summarize the mappings from UML/OCL to other formalism proposed in the past. They are grouped as follows. The first group (G1) includes mappings that *do not not support OCL constraints*. FiniteSAT [19] uses constrained generalization sets for reasoning about finite satisfiability of UML class diagrams. DL [5] encodes the problem of finite model reasoning in UML classes as a constraint satisfaction problem (CSP). MathForm [28] formalizes UML class diagrams using set and partial functions.

The second group (G2) includes mappings that *support OCL constraints, but that do not consider OCL undefinedness*. UMLtoCSP [9] translates UML class diagrams and OCL constraints into CSP. EMFtoCSP [16] is an evolution of UMLtoCSP, which supports EMF models; AuRUS [22, 26] supports verifying and validating UML/OCL conceptual schemes using first-order logic; OCL2FOL [10] also maps UML/OCL class diagrams to first-order logic. OCL-Lite [21] maps a fragment of OCL to DL, ensuring termination. BV-SAT [27] encodes UML/OCL into bit vectors, and solves UML/OCL verification problems based on Boolean satisfiability. PVS [23] and CDOCL-HOL [2] uses higher-order logic: in particular, they map UML/OCL to the specification languages of the theorem provers PVS and Isabelle, respectively. KeY [1] uses dynamic logic, a multi modal extension of first-order logic; Object-Z [25] maps UML/OCL into Object-Z; and finally UML-B [20] maps UML/OCL to the B formal specification.

The third group (G3) includes mappings that *support OCL constraints and consider null-related undefinedness, but not invalid-related undefinedness*. UML2Alloy [3] and USE [15, 18] map UML/OCL to relational logic and use the SAT-based constraint solver KodKod for solving UML/OCL verification problems.

The fourth group (G4) includes mappings that *support OCL constraints and OCL undefinedness*. OCL2MSFOL belongs to this group. OCL-HOL [8] embeds UML/OCL in the specification language of the interactive theorem provers Isabelle. It supports the full OCL language, but it requires advanced user interaction to solve UML/OCL verification problems. OCL2FOL$^+$ [11] maps UML/OCL to first-order logic and uses SMT solvers to attempt to solve automatically UML/OCL verification problems.

Next, we compare more closely OCL2MSFOL with the mappings in groups G3 and G4, and, in particular, with USE and HOL-OCL, from a *practical* point of view. In [14], USE is used to analyse four UML/OCL case studies. For the sake of space limitation, we discuss here only one case study, called *CivilStatus* (the others case studies are similar in size and complexity). The *CivilStatus*' data model contains only one class, Person, with 4 attributes, name, gender (either Male or Female), civilStatus (either Single, Married, Divorced, and Widowed), and spouse, and a self-association, marriage. The *CivilStatus*' invariants are: i) all persons have a defined name, gender, and civilStatus; ii) each person have a unique name; iii) Female-persons are not married with Female-persons, iv) Male-persons are not married with Male-persons, and v) every person has a spouse if and only if his/her civilStatus is Married. The analyses proposed in [14] are of four types: (1) *consistency analysis*, which basically corresponds to satisfiability checking in our setting; (2) *invariants independence analysis*, which in our setting corresponds to checking, for each invariant, that there exists at least a valid instance of the data model in which the rest of the invariants evaluate to true, while the given invariant evaluates to false; (3) *consequence analysis*, which in our setting corresponds to checking that every valid instance of the data model satisfies the given property; and (4) *large state analysis*, which in our setting corresponds to satisfiability checking with additional invariants defining the given "large state". As part of this related work, we have done the four aforementioned analysis using WebOCL2MSFOL with the following results:[6] (1) it proves that the model is consistent

[6]More information can be found at http://software.imdea.

in less that 1 second; (2) it proves that the invariants are independent; each analysis takes less than 1 second; (3) it proves, in less that 1 second, that the model is bigamy-free, i.e., that no person can be married with both Male-person and a Female-person. (5) it proves, in about 6 seconds, that there is a valid instances of the model with 5 Female-persons, 7 Male-persons, and 4 marriage-links. Notice that, in contrast with USE, we prove in (4) that the model is bigamy-free for *all possible instances*, and not only for instances *up to a given size*. This is indeed the key advantage of using SMT-solvers instead of SAT-based constraint solvers for reasoning about UML/OCL models.

With regard to HOL-OCL, there are two significant differences. On the one hand, HOL-OCL uses an interactive theorem prover, Isabelle, for UML/OCL reasoning, which requires advanced knowledge (and possibly time) from the part of the user. OCL2MSFOL, on the contrary, uses SMT solvers with finite model finding capabilities, which, as we have shown, efficiently support automated UML/OCL reasoning. On the other hand, HOL-OCL supports the full OCL language (in fact, it can be considered as providing 'de facto' formal semantics for OCL), while OCL2MSFOL has a number of limitations, as we have discussed before, in supporting the OCL language.

7. CONCLUSION

In this paper we have proposed a mapping, called OCL2-MSFOL, from UML/OCL to many-sorted first-order logic (MSFOL) that supports the direct use of SMT solvers with finite model finding capabilities for automatically reasoning about UML/OCL models. We have also reported on a non-trivial case study, which shows that our mapping is also practical. However, the reader should not forget that our results ultimately depend on the (hard-won) positive logical interaction between (i) our formalization of UML/OCL in MSFOL, and (ii) the heuristics implemented in the SMT solver. This means, in particular, that changes in the SMT solver's heuristics may have consequences (hopefully positive) in the applicability of OCL2MSFOL. It also means that a deeper understanding from our part of the SMT solver's heuristics may lead us to redefine OCL2MSFOL.

8. REFERENCES

[1] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In *FASE 2002, Grenoble, France, Proceedings*, volume 2306 of *LNCS*, pages 327–330. Springer, 2002.

[2] T. Ali, M. Nauman, and M. Alam. An accessible formal specification of the UML and OCL meta-model in isabelle/HOL. In *Multitopic Conference, 2007. INMIC 2007. IEEE.*, pages 1–6. IEEE, 2007.

[3] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *MoDELS 2007, Nashville, USA, Proceedings*, volume 4735 of *LNCS*, pages 436–450. Springer, 2007.

[4] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *SoSyM*, 9(1):69–86, 2010.

org/~dania/tools/ocl2msfol.html

[5] A. Artale, D. Calvanese, and Y. A. Ibáñez-García. Checking full satisfiability of conceptual models. In *DL 2010, Waterloo, Ontario, Canada*, volume 573 of *CEUR Workshop Proceedings*, 2010.

[6] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of the CAV 11.*, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] A. D. Brucker and B. Wolff. HOL-OCL – A Formal Proof Environment for UML/OCL. In *FASE 08*, number 4961 in LNCS, pages 97–100. Springer-Verlag, Heidelberg, 2008.

[8] A. D. Brucker and B. Wolff. HOL-OCL: A formal proof environment for UML/OCL. In *FASE 2008, Budapest, Hungary. Proceedings*, volume 4961 of *LNCS*, pages 97–100. Springer, 2008.

[9] J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *Conference on ASE 2007, Atlanta, Georgia, USA*, pages 547–548. ACM, 2007.

[10] M. Clavel, M. Egea, and M. A. G. de Dios. Checking unsatisfiability for OCL constraints. *Electronic Communications of the EASST*, 24:1–13, 2009.

[11] C. Dania and M. Clavel. OCL2FOL$^+$: Coping with undefinedness. In *OCL@MoDELS*, volume 1092 of *CEUR Workshop Proceedings*, pages 53–62, 2013.

[12] C. Dania and M. Clavel. OCL2MSFOL. definitions, 2016. http: //software.imdea.org/~dania/tools/definitions.pdf.

[13] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[14] M. Gogolla, F. Büttner, and J. Cabot. Initiating a benchmark for UML and OCL analysis tools. In *TAP 2013, Budapest, Hungary. Proceedings*, volume 7942 of *LNCS*, pages 115–132. Springer, 2013.

[15] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *SCP*, 69(1-3):27–34, 2007.

[16] C. A. González, F. Büttner, R. Clarisó, and J. Cabot. EMFtoCSP: A tool for the lightweight verification of EMF models. In *FormSERA*, pages 44–50, 2012.

[17] C. A. González and J. Cabot. Formal verification of static software models in MDE: A systematic review. *IST*, 56(8):821–838, 2014.

[18] 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, *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012. Proceedings*, volume 7590 of *Lecture Notes in Computer Science*, pages 415–431. Springer, 2012.

[19] A. Maraee and M. Balaban. Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In *ECMDA-FA 2007, Haifa, Israel, Proccedings*, volume 4530 of *LNCS*, pages 17–31. Springer, 2007.

[20] R. Marcano-Kamenoff and N. Lévy. Using B formal specifications for analysis and verification of UML/OCL models. In *Workshop on consistency problems in UML-based software development. UML Conference*, Dresden, Germany, 2002.

[21] A. Queralt, A. Artale, D. Calvanese, and E. Teniente. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *DKE*, 73:1–22, 2012.

[22] A. Queralt, G. Rull, E. Teniente, C. Farré, and T. Urpí. AuRUS: Automated reasoning on UML/OCL schemas. In *Conceptual Modeling - ER 2010, Vancouver, BC, Canada. Proceedings*, volume 6412 of *LNCS*, pages 438–444. Springer, 2010.

[23] L. A. Rahim. Mapping from OCL/UML metamodel to PVS metamodel.

[24] A. J. Reynolds. *Finite model finding in satisfiability modulo theories.* PhD thesis, University of Iowa, 2013.

[25] D. Roe, K. Broda, and A. Russo. Mapping UML models incorporating OCL constraints into Object-Z. Technical report, Imperial College of Science, Technology and Medicine, 2003.

[26] G. Rull, C. Farré, A. Queralt, E. Teniente, and T. Urpí. AuRUS: explaining the validation of UML/OCL conceptual schemas. *SoSyM*, 14(2):953–980, 2015.

[27] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using boolean satisfiability. In *DATE 2010, Dresden, Germany.*, pages 1341–1344. IEEE, 2010.

[28] M. Szlenk. Formal semantics and reasoning about UML class diagram. In *DEPCOS-RELCOMEX*, pages 51–59, Washington, DC, USA, 2006. IEEE.

[29] WebOCL2MSFOL Project, 2016. http://actiongui.org/apps/ocl2msfol.

APPENDIX

A. DATA MODELS

Definition 1. A *data model* $\mathcal{D}$ is a tuple $\langle C, CH, AT, AS, ASO, MU\rangle$ such that:

- C is a set of class identifiers.
- CH is a binary relation, $CH \subseteq C \times C$, where $(c, c') \in CH$ represents that c is a subclass of c', also denoted as $c \prec c'$.
- AT is a set of triples $\langle at, c, t\rangle$, also denoted as $at_{(c,t)}$, where at is an attribute identifier, $c \in C$, $t \in C \cup \{\mathsf{Integer}, \mathsf{String}\}$, and c and t are, respectively, the class and the type of the attribute at.
- AS is a set of tuples $\langle as, c, c'\rangle$, also denoted as $as_{(c,c')}$, where as is an association-end identifier, $c, c' \in C$, and c and c' are, respectively, the source and the target classes of as.
- ASO is a symmetric relation, $ASO \subseteq AS \times AS$, where $(as_{(c,c')}, as'_{(c',c)}) \in ASO$ represents that as' is the association-end opposite to as, and vice versa.
- MU is a set of tuples $\langle as_{(c,c')}, mu\rangle$, where $as_{(c,c')} \in AS$, and $mu \in \{0..1, *\}$ represents the multiplicity of the association-end $as_{(c,c')}$.

We assume that data models satisfy the following properties: there is no class whose identifier is Integer or String; attributes and associations-ends have different identifiers; there are no cycles in the class hierarchy; and association-ends are related with exactly another association-end and with exactly one multiplicity.

Definition 2. Let $\mathcal{D}$ be a data model $\langle C, CH, AT, AS, ASO, MU\rangle$. Then, $\mathrm{o2f}_{\mathrm{data}}(\mathcal{D})$ is an MSFOL theory, which is partially defined below. The interested reader can find the full definition of $\mathrm{o2f}_{\mathrm{data}}(\mathcal{D})$ in [12].

- It declares two sorts, *Classifier* and *Type*, to represent the OCL types Classifier and Type. It also declares two sorts, *Int* and *String*, to represent the integer numbers and the strings. [7]
- It declares two constants of sort *Classifier*, *nullClassifier* and *invalClassifier*, to represent, the values null and invalid of type Classifier. In addition, it includes the following axiom:

  $$\neg(\mathit{nullClassifier} = \mathit{invalClassifier}).$$

  Similarly for the type Type.
- It declares two constants of sort *Int*, *nullInt* and *invalInt*, to represent, respectively, the values null and invalid of the primitive data-type Integer. In addition, it includes the following axiom:

  $$\neg(\mathit{nullInt} = \mathit{invalInt}).$$

  Similarly for the primitive data-type String.
- For each class $c \in C$, it declares a predicate c: *Classifier* $\rightarrow$ *Bool*, to represent the objects of type c. In addition, it includes the following axioms:

  $$\forall(x{:}\mathit{Classifier})(c(x) \Rightarrow \neg(\textstyle\bigvee_{c' \in (C\setminus\{c\})} c'(x))).$$
  $$\neg(c(\mathit{nullClassifier}) \vee c(\mathit{invalClassifier})).$$
- For each attribute $at_{(c,\mathsf{Integer})}$, it declares a function at: *Classifier* $\rightarrow$ *Int*. In addition, it includes the following axioms:

  $$\forall(x{:}\mathit{Classifier})((\textstyle\bigvee_{s \preceq c}(s(x))) \Rightarrow at(x) \neq \mathit{InvalInt}).$$
  $$at(\mathit{nullClassifier}) = \mathit{InvalInt}.$$
  $$at(\mathit{invalClassifier}) = \mathit{InvalInt}.$$
- For each association between two classes c and c', with association-ends $as_{(c,c')}$ and $as'_{(c',c)}$, such that $\langle as_{(c,c')}, *\rangle$, $\langle as'_{(c',c)}, *\rangle \in MU$, it declares a predicate as_as': *Classifier* $\times$ *Classifier* $\rightarrow$ *Bool*. In addition, it includes the following axiom:

  $$\forall(x{:}\mathit{Classifier}, y{:}\mathit{Classifier})$$
  $$(as\_as'(x, y) \Rightarrow ((\textstyle\bigvee_{s \preceq c}(s(x))) \wedge (\bigvee_{s' \preceq c'}(s'(y))))).$$
- For each class $c \in C$, it declares a constant c_{type} of sort *Type*. In addition, it includes the following axiom:

  $$\textstyle\bigwedge_{c' \in C\setminus\{c\}} \neg(c_{\mathrm{type}} = c'_{\mathrm{type}}).$$
- It declares two predicates *OclIsTypeOf*, *OclIsKindOf*: *Classifier* $\times$ *Type* $\rightarrow$ *Bool*. In addition, for each class $c \in C$, it includes the following axioms:

  $$\forall(x{:}\mathit{Classifier})$$
  $$(\mathit{OclIsTypeOf}(x, c_{\mathrm{type}}) \Leftrightarrow c(x)).$$

  $$\forall(x{:}\mathit{Classifier})$$
  $$(\mathit{OclIsKindOf}(x, c_{\mathrm{type}}) \Leftrightarrow \textstyle\bigvee_{s \preceq c}(s(x))).$$

[7] We assume that *Int* and *String* are declared with the standard operations and semantics.

B. BOOLEAN EXPRESSIONS

Let *expr* be an expression. In what follows, we assume, without loss of generality, that each iterator in *expr* introduces a different iterator variable. Moreover, we denote by fVars(*expr*) the sequence formed by the free variables in *expr*, sorted alphabetically. Finally, we denote by $\mathrm{App}(P, (x_1, \ldots, x_n), y)$ the atomic formula $P(x_1, ..., x_n, y)$, and we denote by $\mathrm{App}(f, (x_1, \ldots, x_n))$ the term $f(x_1, ..., x_n)$.

Definition 3. Let *expr* be an expression. Then,

$\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}.\mathsf{oclIsUndefined}()) =$
$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}).$

$\mathrm{o2f}_{\mathrm{false}}(\mathit{expr}.\mathsf{oclIsUndefined}()) =$
$\neg(\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr})).$

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}.\mathsf{oclIsUndefined}()) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}.\mathsf{oclIsUndefined}()) = \bot.$

Definition 4. Let *expr* be an expression of the appropriate type. Then,

$\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}{\rightarrow}\mathsf{notEmpty}()) =$
$\exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}), \mathrm{fVars}(\mathit{expr}), x))$
$\wedge\ \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr})).$

$\mathrm{o2f}_{\mathrm{false}}(\mathit{expr}{\rightarrow}\mathsf{notEmpty}()) =$
$\forall(x)(\neg(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}), \mathrm{fVars}(\mathit{expr}), x))$
$\wedge\ \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr})).$

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}{\rightarrow}\mathsf{notEmpty}()) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}{\rightarrow}\mathsf{notEmpty}()) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}).$

Definition 5. Let *src* and *body* be expressions of the appropriate types. Then,

$\mathrm{o2f}_{\mathrm{true}}(\mathit{src}{\rightarrow}\mathsf{forAll}(x \mid \mathit{body})) =$
$\forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \Rightarrow \mathrm{o2f}_{\mathrm{true}}(\mathit{body}))$
$\wedge\neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src})).$

$\mathrm{o2f}_{\mathrm{false}}(\mathit{src}{\rightarrow}\mathsf{forAll}(x \mid \mathit{body})) =$
$\exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge \mathrm{o2f}_{\mathrm{false}}(\mathit{body}))$
$\wedge\neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src})).$

$\mathrm{o2f}_{\mathrm{null}}(\mathit{src}{\rightarrow}\mathsf{forAll}(x \mid \mathit{body})) =$
$\neg\ \mathrm{o2f}_{\mathrm{inval}}(\mathit{src})$
$\wedge\ \exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge \mathrm{o2f}_{\mathrm{null}}(\mathit{body}))$
$\wedge\ \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x)$
$\Rightarrow (\mathrm{o2f}_{\mathrm{true}}(\mathit{body}) \vee\ \mathrm{o2f}_{\mathrm{null}}(\mathit{body}).$

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}{\rightarrow}\mathsf{forAll}(x \mid \mathit{body})) =$
$\mathrm{o2f}_{\mathrm{inval}}(\mathit{src})$
$\vee\ \exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge \mathrm{o2f}_{\mathrm{inval}}(\mathit{body}))$
$\wedge\ \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x)$
$\Rightarrow (\mathrm{o2f}_{\mathrm{true}}(\mathit{body}) \vee \mathrm{o2f}_{\mathrm{null}}(\mathit{body}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{body}))).$

C. NON-BOOLEAN EXPRESSIONS

In what follows, for t a type, $\mathrm{NullOf}(t) = \mathit{nullClassifier}$ if t is a class type, $\mathrm{NullOf}(t) = \mathit{nullInt}$ if $t = \mathsf{Integer}$, and $\mathrm{NullOf}(t) = \mathit{nullString}$ if $t = \mathsf{String}$. Similarly, $\mathrm{InvalOf}(t) = \mathit{invalInt}$ if t is a class type, $\mathrm{InvalOf}(t) = \mathit{invalInt}$ if $t = \mathsf{Integer}$, and $\mathrm{InvalOf}(t) = \mathit{invalString}$ if $t = \mathsf{String}$.

Definition 6. Let v be a variable of type t. Then,

$\mathrm{o2f}_{\mathrm{eval}}(v) = v.$

$\mathrm{o2f}_{\mathrm{null}}(v) = (v = \mathrm{NullOf}(t)).$

$\mathrm{o2f}_{\mathrm{inval}}(v) = (v = \mathrm{InvalOf}(t)).$

Definition 7. Let i be an integer. Then,

$\mathrm{o2f}_{\mathrm{eval}}(i) = i.$

$\mathrm{o2f}_{\mathrm{null}}(i) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(i) = \bot.$

Definition 8. Let *at* be an attribute of type t and let *expr* be an expression of the appropriate type. Then,

$\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}.\mathit{at}) = \mathit{at}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr})).$

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}.\mathit{at}) = (\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}.\mathit{at}) = \mathrm{NullOf}(t)).$

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}.\mathit{at}) = \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}).$

Definition 9. Let c be a class. Then,

$\mathrm{o2f}_{\mathrm{eval}}(c.\mathsf{allInstances}()) = [c].$

$\mathrm{o2f}_{\mathrm{def\_c}}(c.\mathsf{allInstances}()) =$
$\{\forall(x)(\mathrm{App}([c], \emptyset, x) \Leftrightarrow (\bigvee_{s \preceq c}(s(x))))\}.$

$\mathrm{o2f}_{\mathrm{null}}(c.\mathsf{allInstances}()) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(c.\mathsf{allInstances}()) = \bot.$

Definition 10. Let *src* and *body* be expressions of the appropriate types. Then,

$\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}{\rightarrow}\mathsf{select}(x|\mathit{body})) = [\mathit{src}{\rightarrow}\mathsf{select}(x|\mathit{body})]$

and

$\mathrm{o2f}_{\mathrm{def\_c}}(\mathit{src}{\rightarrow}\mathsf{select}(x|\mathit{body})) =$
$\{\forall(\vec{y})(\forall(x)(\mathrm{App}([\mathit{src}{\rightarrow}\mathsf{select}(x|\mathit{body})], \vec{y}, x)$
$\Leftrightarrow$
$\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge \mathrm{o2f}_{\mathrm{true}}(\mathit{body})))\}.$

where $\vec{y} = \mathrm{fVars}(\mathit{src}{\rightarrow}\mathsf{select}(x \mid \mathit{body}))$.

$\mathrm{o2f}_{\mathrm{null}}(\mathit{src}{\rightarrow}\mathsf{select}(x|\mathit{body})) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}{\rightarrow}\mathsf{select}(x|\mathit{body})) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{src}).$