Proceedings of the Workshop
The Pragmatics of OCL and Other Textual Specification Languages
at MoDELS 2009

Checking Unsatisfiability for OCL Constraints

Manuel Clavel, Marina Egea and Miguel A. García de Dios

13 pages

Checking Unsatisfiability for OCL Constraints

Manuel Clavel[1], Marina Egea[2] and Miguel A. García de Dios[3]*

[1] manuel.clavel@imdea.org
[3] miguelangel.garcia@imdea.org
IMDEA Software Institute, Madrid, Spain

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain

[2] marinae@inf.ethz.ch
Information Security / ZISC
ETH Zürich, Switzerland

Abstract: In this paper we propose a mapping from a subset of OCL into first-order logic (FOL) and use this mapping for checking the unsatisfiability of sets of OCL constraints. Although still preliminary work, we argue in this paper that our mapping is both simple, since the resulting FOL sentences closely mirror the original OCL constraints, and practical, since we can use automated reasoning tools, such as automated theorem provers and SMT solvers to automatically check the unsatisfiability of non-trivial sets of OCL constraints.

Keywords: OCL, unsatisfiability, automated deduction, SMT solvers

1 Motivation

The lack of tool support for OCL was pointed out in [CBC05] as a main cause for the limited adoption of the language in industry. Since then, many initiatives have been brought to fruition and their outcomes are available to designers (see [Dem05]). Among the tool categories that have received significant attention are:

- *Parsers*: to check the syntactical well-formedness of an expression: e.g., the Dresden OCL 2.0 parser [Kon05, Sof07].

- *Evaluators*: to obtain the value of an expression within a contextual model: e.g., USE [Dat06], MDT OCL [Hus08], and EOS [CED08, DCE08].

- *Translators*: to map (for different purposes) an expression into a (logically equivalent) expression in other languages and/or formalisms: e.g.,

  - OCL2SQL [Hei06, Sof07] maps OCL constraints into SQL queries;

- UMLtoCSP [CCR07b, CCR07a] maps OCL constraints into constraint programming expressions to support automated bounded verification of UML class diagrams annotated with OCL constraints;

- MOMENT [CRBG08] and ITP-OCL [CE08] map OCL into equational logic (although using different approaches) to support automated evaluation of OCL expressions using term-rewriting.

- KeY [BS07] includes a mapping of OCL into first-order logic to allow interactive reasoning about UML diagrams with OCL constraints.

- HOL-OCL [Bru07, BW07] maps OCL into higher-order logic also to allow interactive reasoning about UML diagrams with OCL constraints.

The work presented here belongs to the third category: it proposes a mapping from OCL to first-order logic, which is defined with the purpose of supporting (*unbounded*) unsatisfiability checks for OCL expressions using *automated* reasoning tools. In our view, being able to check the *unsatisfiability* of (sets of) OCL expressions is a powerful tool, since it will allow modelers to (among other tasks):

- Verify class invariants, by checking that they logically imply the expected constraints/properties;

- Verify method preconditions, by checking that the class invariants do not logically imply their negations; and

- Verify method postconditions, by checking that they do not logically imply the negation of (any of) the class invariants.

However, also in our view, what will make an unsatisfiability checker not only powerful, but also practical, is being *automated*. Given the undecidable nature of the full OCL language, one can only expect to have an automated unsatisfiability checker for a large class of OCL expressions. In this paper we do not attempt to define how large and/or interesting is the class of unsatisfiable OCL expressions that we are able to check automatically. Nevertheless, we will argue that our mapping is both simple, since the resulting FOL sentences closely mirror the original OCL constraints, and practical, since we can use existing automated theorem provers (e.g., Prover9 [McC06]) and/or SMT solvers (e.g., Yices [DM08]) to automatically check the unsatisfiability of non-trivial sets of OCL constraints.

Organization In Section 2 we define our notion of *unsatisfiability* for OCL constraints. Then, in Section 3 we define our mapping from OCL to FOL. Next, in Section 4 we report on our experience using two different automated reasoning tools for checking the unsatisfiability of (sets of) OCL constraints: namely, Prover9 [McC06] (an automated theorem prover) and Yices [DM08] (an SMT solver). We conclude with a discussion on related and future work.
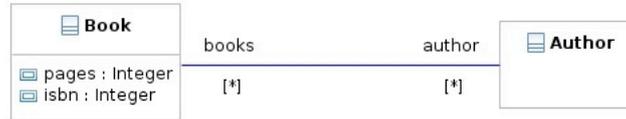
Figure 1: A simple *Library*-model.

2 Unsatisfiability of OCL constraints

In this section we introduce our notion of *unsatisfiability* for OCL constraints, as well as the examples that we will use in the following sections. The notion of *unsatisfiability* that we propose emphasizes the logical meaning of OCL constraints; in fact, it basically translates to OCL the standard notion of unsatisfiability for logic formulas. There are other notions of satisfiability/unsatisfiability for OCL constraints in the literature, which we will briefly discuss at the end of this section.

In what follows, we denote by OCL *constraint* any OCL expression of type Boolean. We do not assume that instances of models always have a finite a number of elements.

Definition 1 Given a model (class diagram) $\mathcal{M}$, and a set of OCL constraints Φ, we say that Φ is $\mathcal{M}$-*unsatisfiable* if and only if there does not exist an $\mathcal{M}$-instance (object diagram) $\mathcal{O}$ on which every constraint in Φ evaluates to true.

To illustrate this notion, we introduce the following example. Table 1 shows a list of OCL constraints: they all refer to the simple class diagram *Library* shown in Figure 1. In this model, libraries contains *Books* and books have *Authors*, *pages*, and an *ISBN* code.

According to Definition 1, the following subsets (among others) of the constraints shown in Table 1 are *Library*-unsatisfiable: {1,2}, {1,8}, {1,10}, {2,3}, {2,4}, {2,5}, {2,6}, {7,8}, {11, 13}, {12}, {14}, and {15}.

Notice that the subset {9,10,11} is not *Library*-unsatisfiable: a library with just one book will satisfy these constraints. On the other hand, the subset {9,10,11,16} is indeed *Library*-unsatisfiable. Notice also that the subset {17} is not *Library*-unsatisfiable: a library with an infinite number of books will satisfy this constraint. On the other hand, the subset {17, 18} is indeed *Library*- unsatisfiable.

As mentioned before, other notions of satisfiability/unsatisfiability of UML models with OCL constraints can be found in the literature. In particular, the notions used in [CCR07b, CT07] are those of *weak* and *strong satisfiability* (and related notions are also introduced in [BW07]). Weak satisfiability means that there exists a finite instance of the model in which at least one class is populated with at least one element. Strong satisfiability means that there exists a finite instance of the model in which all its classes are populated with at least one element. Notice that, if a set of constraints is unsatisfiable (in our sense), then it can not be weak nor strong satisfiable. On the other hand, if a set of constraints is not weak nor strong satisfiable, it does not imply that is unsatisfiable (in our sense).

1. Book.allInstances()−>isEmpty().

2. Book.allInstances()−>exists(x|x.pages > 300).

3. Book.allInstances()−>forAll(x|x.pages < 300).

4. Book.allInstances()−>select(x|x.pages > 300)−>isEmpty().

5. Book.allInstances()−>reject(x|x.pages <= 300)−>isEmpty().

6. Book.allInstances()−>collect(x|x.pages)−>asSet()−>forAll(i|i < 300).

7. Book.allInstances()−>forAll(x|x.author−>isEmpty()).

8. Author.allInstances()−>exists(a|a.books−>notEmpty()).

9. Book.allInstances()−>forAll(x,y|x<>y implies x.isbn<>y.isbn)).

10. Book.allInstances()−>exists(x|Book.allInstances()−>excluding(x)
    −>forAll(y|y.isbn=x.isbn)).

11. Book.allInstances()−>notEmpty().

12. Book.allInstances()−>exists(x|Book.allInstances()−>excludes(x)).

13. Book.allInstances()−>forAll(x|Book.allInstances()−>excluding(x)−>includes(x)).

14. Book.allInstances()−>exists(x|Book.allInstances()−>excluding(x)−>includes(x)).

15. Book.allInstances()−>collect(x|x.author)−>asSet()−>exists(y|y.books
    −>isEmpty())).

16. Book.allInstances()−>size() > 1.

17. Book.allInstances()−>forAll(b|Book.allInstances()−>exists(x|x.pages > b.pages)).

18. Book.allInstances()−>size() = 2.

Table 1: List of constraints

3 A mapping from OCL to FOL

In this section we define a mapping from a subset of OCL into first-order logic (FOL). Given a set of OCL constraints, our mapping generates a set of FOL formulas such that, if the resulting set is unsatisfiable, then the original set is also unsatisfiable. We will argue in Section 4 that our mapping is both simple and practical. At this preliminary stage, we are not ready, however, to provide a formal proof of the *correctness* of this mapping (with respect to a well-defined formal semantics for OCL).

OCL constraints specify properties that must be satisfied by a model. In order to do so in a concise way, OCL provides different constructors to refer to specific collections of elements. In a nutshell, our mapping is defined recursively over the structure of OCL expressions:

- Boolean-expressions are translated to formulas, which essentially mirror their logical structure; Integer-expressions are basically copied; at this point, we do not consider String-expressions.

- Collection-expressions are translated to predicates, whose meaning is defined by additional formulas generated by the mapping; at this point, we only consider Set-expressions.

- Association-ends are translated by predicates, which are also defined by formulas generated by the mapping; at this point, we do not consider qualified associations.

- Attributes are translated by functions, which are left undefined by the mapping.

The function *map*() below defines our mapping from OCL to FOL. We do not attempt to cover here the full OCL language, but only a subset that is significantly enough so as to show the potential of our proposal.

In what follows, given an iterator variable x, the expression $x^\flat$ denotes a fresh new logical variable. Similarly, given Boolean-expressions *BoolExpr*, object expressions *ObjExpr*, and Set-expressions *SetExpr* and *SetExpr'*, the expressions [collect, *SetExpr*, *SetExpr'*]$^\flat$, [select, *SetExpr*, *BoolExpr*]$^\flat$, [reject, *SetExpr*, *BoolExpr*]$^\flat$, [including, *SetExpr*, *ObjExpr*]$^\flat$, and [excluding, *SetExpr*, *ObjExpr*]$^\flat$ denote fresh new predicate names.

The auxiliary function *name*(), used in the definition of *map*(), is the one in charge of providing unique names for the FOL predicates that translate the different OCL Collection-expressions; it also translates OCL literal values by the corresponding FOL terms.

Definition 2   The auxiliary function name() is defined by the following clauses:

$$
\begin{aligned}
\text{name}(Integer) &= Integer. \\
\text{name}(-Integer) &= -Integer. \\
\text{name}(Integer[+\,|\,*]Integer') &= Integer[+\,|\,\times]Integer'. \\
\text{name}(Var) &= Var. \\
\text{name}(ClassId) &= ClassId. \\
\text{name}(ObjExpr.Attr) &= Attr(\text{name}(ObjExpr)). \\
\text{name}(ObjExpr.AssocEnd) &= AssocEnd(\text{name}(ObjExpr)). \\
\text{name}(ClassExpr.\text{allInstances}()) &= \text{name}(ClassExpr). \\
\text{name}(SetExpr{-}{>}\text{collect}(x|SetExpr')) &= [\text{collect}, SetExpr, SetExpr']^\flat. \\
\text{name}(SetExpr{-}{>}\text{select}(x|BoolExpr)) &= [\text{select}, SetExpr, BoolExpr]^\flat. \\
\text{name}(SetExpr{-}{>}\text{reject}(x|BoolExpr)) &= [\text{reject}, SetExpr, BoolExpr]^\flat. \\
\text{name}(SetExpr{-}{>}\text{excluding}(x|ObjExpr)) &= [\text{excluding}, SetExpr, ObjExpr]^\flat. \\
\text{name}(SetExpr{-}{>}\text{including}(x|ObjExpr)) &= [\text{including}, SetExpr, ObjExpr]^\flat.
\end{aligned}
$$

The auxiliary function *in_coll*(), also used in the definition of *map*(), basically returns the atomic formula that represents the application of a given predicate to a given number of arguments.

Definition 3   The auxiliary function in_coll() is defined by the following clause:

$$in\_coll(Name, x) \quad = Name(x).$$

Finally, the auxiliary function *make_conj* returns the conjunction of a given set of formulas.

Definition 4   The auxiliary function make_conj() is defined by the following clauses:

$$
\begin{aligned}
make\_conj(\emptyset) &= \top. \\
make\_conj(\{\phi\}) &= \phi. \\
make\_conj(\{\phi_1, \ldots, \phi_{n+1}\}) &= \phi_1 \wedge \cdots \wedge \phi_{n+1}.
\end{aligned}
$$

We are now ready to define our mapping from OCL to FOL. First, the function *map()* generates, by default, the sentences defining the predicates that represent, in our mapping, the association-ends specified in the given model.

Definition 5   Given an association between two classes $Class_1$ and $Class_2$, with association-ends $AssocEnd^{Class_1}$ and $AssocEnd^{Class_2}$, the function map() generates, by default, the following sentences:

$$
\begin{aligned}
&\forall (x, y)(AssocEnd^{Class_1}(x, y) \Rightarrow Class_1(y)). \\
&\forall (x, y)(AssocEnd^{Class_2}(x, y) \Rightarrow Class_2(y)). \\
&\forall (x, y)(AssocEnd^{Class_1}(x, y) \Leftrightarrow AssocEnd^{Class_2}(y, x)).
\end{aligned}
$$

Next, we define the mapping from OCL Boolean-expressions to FOL formulas: essentially, we mirror the logical structure of the OCL expressions in the resulting FOL formulas. In particular, we map iterator variables into logical variables, which are existentially or universally quantified depending on the iterator used.

Definition 6   The function *map()* on Boolean-expressions is defined by clauses shown in Figure 2.

Finally, we define the mapping from OCL Collection-expressions to FOL formulas. Recall that collections are represented in our mapping by predicates. The function *map()* defines these predicates by generating the appropriate FOL formulas. Recall also that the only Collection-expressions currently covered by our mapping are the Set-expressions.

Definition 7   The function *map()* on Collection-expressions is defined by clauses shown in Figure 3.

4   Examples

In this section, we argue, using a set of examples, that our mapping is both simple and practical.

map(true)
  = {⊤}.
map(false)
  = {⊥}.
map(*IntExpr*[> | < | >= | <= | = | <>]*IntExpr′*)
  = {name(*IntExpr*)[>|>|>=|<=|=|<>]name(*IntExpr′*)}.
map(not *BoolExpr*)
  = {¬(make_conj(map(*BoolExpr*)))}.
map(*BoolExpr* and *BoolExpr′*)
  = {make_conj(map(*BoolExpr*)) ∧ make_conj(map(*BoolExpr′*))}.
map(*BoolExpr* or *BoolExpr′*)
  = {make_conj(map(*BoolExpr*)) ∨ make_conj(map(*BoolExpr′*))}.
map(*BoolExpr* implies *BoolExpr′*)
  = {make_conj(map(*BoolExpr*)) ⇒ make_conj(map(*BoolExpr′*))}.
map(*SetExpr*−>isEmpty())
  = {∀($x^♭$)(¬(in_coll(name(*SetExpr*), $x^♭$)))} ∪ map(*SetExpr*).
map(*SetExpr*−>notEmpty())
  = {∃($x^♭$)(in_coll(name(*SetExpr*), $x^♭$))} ∪ map(*SetExpr*).
map(*SetExpr*−>excludes(*ObjExpr*))
  = {¬(in_coll(name(*SetExpr*), name(*ObjExpr*)))} ∪ map(*SetExpr*).
map(*SetExpr*−>includes(*ObjExpr*))
  = {in_coll(name(*SetExpr*), name(*ObjExpr*))} ∪ map(*SetExpr*).
map(*SetExpr*−>exists(x|*BoolExpr*))
  = {∃($x^♭$)(in_coll(name(*SetExpr*), $x^♭$) ∧ make_conj(map(*BoolExpr*[$x ↦ x^♭$])))}
    ∪ map(*SetExpr*).
map(*SetExpr*−>forAll(x|*BoolExpr*))
  = {∀($x^♭$)(in_coll(name(*SetExpr*), $x^♭$) ⇒ make_conj(map(*BoolExpr*[$x ↦ x^♭$])))}
    ∪ map(*SetExpr*).

Figure 2: Definition: *map*() on Boolean-expressions.

$\text{map}(\textit{ClassId}.\text{allInstances}())$
$\quad = \emptyset.$
$\text{map}(\textit{SetExpr}-\!\!>\!\text{collect}(x|\textit{SetExpr}')-\!\!>\!\text{asSet}())$
$\quad = \{\forall(y^\flat)(\text{in\_coll}(\text{name}(\textit{SetExpr}-\!\!>\!\text{collect}(x|\textit{SetExpr}'),y^\flat)$
$\quad\quad\quad \Leftrightarrow \exists(w^\flat)(\text{in\_coll}(\text{name}(\textit{SetExpr}),w^\flat)\wedge$
$\quad\quad\quad\quad\quad \text{in\_coll}(\text{name}(\textit{SetExpr}'[x\mapsto w^\flat]),y^\flat)))) \}.$
$\text{map}(\textit{SetExpr}-\!\!>\!\text{collect}(x|\textit{ObjExpr})-\!\!>\!\text{asSet}())$
$\quad = \{\forall(y^\flat)(\text{in\_coll}(\text{name}(\textit{SetExpr}-\!\!>\!\text{collect}(x|\textit{ObjExpr})),y^\flat)$
$\quad\quad\quad \Leftrightarrow \exists(w^\flat)(\text{in\_coll}(\text{name}(\textit{SetExpr}),w^\flat)\wedge$
$\quad\quad\quad\quad\quad y^\flat = \text{name}(\textit{ObjExpr}[x\mapsto w^\flat])))) \}.$
$\text{map}(\textit{SetExpr}-\!\!>\!\text{select}(x|\textit{BoolExpr}))$
$\quad = \{\forall(y^\flat)(\text{in\_coll}(\text{name}(\textit{SetExpr}-\!\!>\!\text{select}(x|\textit{BoolExpr}[x\mapsto y^\flat])),y^\flat)$
$\quad\quad\quad \Leftrightarrow (\text{in\_coll}(\textit{name}(\textit{SetExpr}),y^\flat)\wedge$
$\quad\quad\quad\quad \text{make\_conj}(\text{map}(\textit{BoolExpr}[x\mapsto y^\flat]))))) \}.$
$\text{map}(\textit{SetExpr}-\!\!>\!\text{reject}(x|\textit{BoolExpr}))$
$\quad = \{\forall(y^\flat)(\text{in\_coll}(\text{name}(\textit{SetExpr}-\!\!>\!\text{reject}(x|\textit{BoolExpr}[x\mapsto y^\flat])),y^\flat)$
$\quad\quad\quad \Leftrightarrow (\text{in\_coll}(\text{name}(\textit{SetExpr}),y^\flat)\wedge$
$\quad\quad\quad\quad \neg(\text{make\_conj}(\text{map}(\textit{BoolExpr}[x\mapsto y^\flat])))))) \}.$
$\text{map}(\textit{SetExpr}-\!\!>\!\text{excluding}(\textit{ObjExpr}))$
$\quad = \{\forall(y^\flat)(\text{in\_coll}(\text{name}(\textit{SetExpr}-\!\!>\!\text{excluding}(\textit{ObjExpr})),y^\flat)$
$\quad\quad\quad \Leftrightarrow (\text{in\_coll}(\text{name}(\textit{SetExpr}),y^\flat) \wedge y^\flat \neq \text{name}(\textit{ObjExpr}))) \}.$
$\text{map}(\textit{SetExpr}-\!\!>\!\text{including}(\textit{ObjExpr}))$
$\quad = \{\forall(y^\flat)(\text{in\_coll}(\text{name}(\textit{SetExpr}-\!\!>\!\text{including}(\textit{ObjExpr})),y^\flat)$
$\quad\quad\quad \Leftrightarrow (\text{in\_coll}(\text{name}(\textit{SetExpr}),y^\flat) \vee y^\flat = \text{name}(\textit{ObjExpr}))) \}.$

Figure 3: Definition: *map*() on Collection-expressions.

.

4.1 Mapping constraints

With respect to other mappings previously proposed, one advantage of our mapping is that, in addition to be mechanizable, the resulting formulas are similar to the original OCL constraints, both in their size and in their logical structure. In this sense, we claim that our mapping is *simple*. To illustrate this claim, we show in Figure 4 the FOL formulas that the function *map*() generates to define the predicates that represent, in our mapping, the two association-ends in the *Library*-model. More interestingly, we show in Figure 5 the FOL formulas resulting from applying the function *map*() to a representative subset of the constraints listed in Table 1.

$$\forall (x,y)(\mathrm{books}(x,y) \Rightarrow \mathrm{Book}(y))$$
$$\forall (x,y)(\mathrm{author}(x,y) \Rightarrow \mathrm{Author}(y))$$
$$\forall (x,y)(\mathrm{books}(x,y) \Leftrightarrow \mathrm{author}(y,x))$$

Figure 4: Example: Mapping association-ends.

4.2 Checking unsatisfiability

Another crucial advantage of our mapping is that the resulting formulas can be checked for unsatisfiability using *automated* reasoning tools. To illustrate our point, we have tried to automatically prove the unsatisfiability of different subsets of the constraints shown in Table 1 using Prover9 [McC06] (an automated theorem prover) and Yices [DM08] (an SMT solver). The results are shown in Table 2. Both tools finished their tasks in less than a second, running on a standard laptop computer. In our experiments, we used both tools with their default (command) options.

Prover9 [McC06] is a resolution/paramodulation automated theorem prover for first-order and equational logic. It uses two default limits which, although good in practice, can prevent proofs from being found. Not surprisingly (since it does not support integer arithmetic), Prover9 could not automatically prove the unsatisfiability of some of the subsets of constraints in our experiment.

Yices [DM08] is a high-performance SMT solver that decides the satisfiability of propositional formulas that mix uninterpreted function symbols and equality with interpreted symbols from various theories, in particular for linear real and integer arithmetic, but also for recursive datatypes, tuples, records, lambda expressions and quantifiers among others. Although Yices is not complete when quantifiers are used, it was able to automatically prove the unsatisfiability of all the subsets of constraints in our experiment. This result is certainly encouraging for our purposes; moreover, we expect to take advantage of its decision procedures for recursive datatypes, tuples, and records, in order to prove the unsatisfiability of more complex subsets of OCL constraints.

(1)   map(Book.allInstances()−>isEmpty())

$$= \{\forall(x)(\neg \text{Book}(x))\}.$$

(2)   map(Book.allInstances()−>exists(x|x.pages > 300))

$$= \{\exists(x)(\text{Book}(x) \wedge (\text{pages}(x) > 300))\}.$$

(3)   map(Book.allInstances()−>forAll(x | x.pages < 300))

$$= \{\forall(x)(\text{Book}(x) \Rightarrow (\text{pages}(x) < 300))\}.$$

(4)   map(Book.allInstances()−>select(x|x.pages > 300)−>isEmpty())

$$= \{\forall(x)\neg(\text{select1}(x)),$$
$$\forall(x)(\text{select1}(x) \Leftrightarrow (\text{Book}(x) \wedge (\text{pages}(x) > 300)))\}.$$

(7)   map(Book.allInstances()−>forAll(x|x.author−>isEmpty()))

$$= \{\forall(x)(\text{Book}(x) \Rightarrow \forall(y)(\neg(\text{author}(x,y))\}.$$

(8)   map(Author.allInstances()−>exists(a|a.books−>notEmpty()))

$$= \{\exists(a)(\text{Author}(a) \wedge \exists(x)(\text{books}(a,x)))\}.$$

(9)   map(Book.allInstances()−>forAll(x,y|x<>y implies x.isbn<>y.isbn))

$$= \{\forall(x)(\text{Book}(x) \Rightarrow \forall(y)(\text{Book}(y) \Rightarrow (x \neq y \Rightarrow \text{isbn}(x) \neq \text{isbn}(y))))\}.$$

(10)   map(Book.allInstances()−>exists(x|Book.allInstances()−>excluding(x)
                          −>forAll(y|y.isbn=x.isbn)))

$$= \{\exists(x)(\text{Book}(x) \wedge [\forall(y)(\text{excluding1}(y) \Rightarrow (\text{isbn}(x) = \text{isbn}(y)))$$
$$\wedge \forall(z)(\text{excluding1}(z) \Leftrightarrow (\text{Book}(z) \wedge z \neq x))])\}.$$

(11)   map(Book.allInstances()−>notEmpty())

$$= \{\exists(x)(\text{Book}(x))\}.$$

(12)   map(Book.allInstances()−>exists(x|Book.allInstances()−>excludes(x)))

$$= \{\exists(x)(\text{Book}(x) \wedge \neg(\text{Book}(x)))\}.$$

(13)   map(Book.allInstances()−>forAll(x|Book.allInstances()−>excluding(x)
                          −>includes(x)))

$$= \{\forall(x)((\text{Book}(x) \Rightarrow \text{excluding1}(x))$$
$$\wedge \forall(y)((\text{Book}(y) \wedge y \neq x) \Leftrightarrow \text{excluding1}(y)))\}.$$

(14)   map(Book.allInstances()−>exists(x|Book.allInstances()−>excluding(x)
                          −>includes(x)))

$$= \{\exists(x)(\text{Book}(x) \wedge \text{excluding1}(x)$$
$$\wedge (\forall(y)((\text{Book}(y) \wedge y \neq x) \Leftrightarrow \text{excluding1}(y))))\}.$$

(15)   map(Book.allInstances()−>collect(x|x.author)−>asSet()
      −>exists(y|y.books−>isEmpty()))

$$= \{\exists(y)(\text{collect1}(y) \wedge \forall(x)(\neg(\text{books}(y,x))),$$
$$\forall(z)(\text{collect1}(z) \Leftrightarrow \exists(w)(\text{Book}(w) \wedge \text{author}(w,z))))\}.$$

Figure 5: Example: Mapping constraints.

5 Related work

As already mentioned, there are other mappings from OCL into different languages and/or formalisms, each one with its own purposes and target reasoning tool, which makes difficult to do a general comparison. We discuss here only those proposals that support (in one way or another) our same objectives, namely, checking unsatisfiability for OCL constraints. In a nutshell, none of these mappings supports both *unbounded* and *automated* unsatisfiability checks for OCL constraints, as our mapping does, for a significantly subset of the language.

	{1,2}	{1,8}	{1,10}	{2,3}	{2,4}	{2,5}	{2,6}	{7,8}	{11,13}	{12}	{14}	{15}
Prover9	√	√	√	•	√	•	•	√	√	√	√	√
Yices	√	√	√	√	√	√	√	√	√	√	√	√

Table 2: Case study: checking unsatisfiability

The *KeY tool* [BS07] is able to generate proof obligations from checking different properties of UML models with OCL constraints (e.g., invariant preservation), using a translation of OCL into first order predicate logic. To the best of our knowledge, the KeY tool translates OCL collection expressions into first order terms, introducing additional axioms in order to constrain the interpretation of the new function symbols which represent the OCL collections [BKS02].[1] Users can then *interact* with the KeY theorem prover to logically reason about their UML models. A limited amount of automated reasoning is provided, which is far from what is currently offered by modern SMT solvers.

HOL-OCL [BW07] is an *interactive* proof environment for OCL. It is implemented as a shallow embedding of OCL into higher-order logic (HOL) within the theorem prover Isabelle. The resulting translations may be hard to understand by standard software engineers. Also, since the amount of automated reasoning which is supported by HOL-OCL is limited, software engineers may find hard to use this tool when reasoning about their UML models.

UMLtoCSP [CCR07b] provides *bounded* automatic verification of UML models annotated with OCL constraints. The users must limit the search space by explicitly indicating the number of objects in each class, the number of links of each association and the possible values of each attribute. When the tool can not find a satisfying instance within the specified search space, this does not means that the property does not hold: it can still hold for values outside the search space (and the user may try to verify the property with wider intervals).

UML2Alloy [Ana, ABGR07] is a front-end that transforms UML diagrams annotated with OCL constraints into the Alloy notation. It translates the model into a Boolean expression, which is then analysed by the SAT solvers implemented within Alloy. As in the case of UMLtoCSP, the domain must be *bounded* by the user before analysing the model.

6 Conclusions and future work

In this paper we have proposed a mapping from a significant subset of OCL into first-order logic (FOL). Although this is still preliminary work, we have argued that our mapping is both simple, since the resulting FOL sentences closely mirror the original OCL constraints, and practical, since we can use automated theorem provers (e.g., Prover9) and/or SMT solvers (e.g., Yices) to automatically check the unsatisfiability of non-trivial sets of OCL constraints.

In the near future, we plan to extend our mapping to deal with larger subsets of OCL. Since this is preliminary work, the list of currently missing features is certainly large: we discuss here

---

[1] Since the result is often lengthy and hard to read, it was suggested (via some examples) in [BKS02] to use, as an alternative, a predicative translation to first order formulas. Again, to the best of our knowledge, this line of research has not been followed up within the KeY community. Also the examples shown in [BKS02] do not provide enough information so as to understand how this predicative translation will deal with other cases, like collect-expressions, and including|excluding-expressions, or with nested-iterator expressions.

only the most important ones. First, we should be able to deal with bags. Our idea here is to translate Bag-expressions using predicates, as we do for Set-expressions, but with an additional argument indicating the number of occurrences of a given element in the bag. Of course, the proposed mapping for the different Collect-operations will have to be modified accordingly. Second, we should be able to deal with generalizations. The idea here is to add, by default, the expected sentences formalizing that every element in the subclass-collection also belongs to the super-class collection. Third, we should be able to deal with size-expressions. Here, we do not have yet a general solution: dealing with constraints like (16) and (18) in Table 1 and, in general, with constraints that simply restrict the multiplicity of collections, is rather simple; defining a mapping for arbitrary size-expressions appearing anywhere inside constraints requires further investigation. Finally, it would be interesting to define a mapping for general Collection-expressions and for Tuple-expressions, and to explore the capabilities of SMT solvers to automatically check the unsatisfiability of the resulting formulas. In the long term, we should prove, of course, the correctness of our mapping with respect to a formal semantics of the OCL language.

Bibliography

[ABGR07] K. Anastasakis, B. Bordbar, G. Georg, I. Ray. UML2Alloy: A Challenging Model Transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*. LNCS 4735. 2007.

[Ana]   K. Anastasakis. UML2Alloy. http://www.cs.bham.ac.uk/~bxb/UML2Alloy.

[BKS02]   B. Beckert, U. Keller, P. H. Schmitt. Translating the Object Constraint Language into First-order Predicate Logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*. 2002.

[Bru07]   A. D. Brucker. *An Interactive Proof Environment for Object Oriented Specifications*. PhD thesis, ETH Zurich, 2007.

[BW07]   A. D. Brucker, B. Wolff. The HOL-OCL tool. http://www.brucker.ch/, 2007. ETH Zurich.

[BS07]   B. Beckert, R. Hähnle, P. H. Schmitt (eds.). *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[CBC05]   D. Chiorean, M. Bortes, D. Corutiu. Proposals for a Widespread Use of OCL. In *Tool Support for OCL and Related Formalisms - Needs and Trends– MoDELS'05 Conference Workshop*. Pp. 68–82. 2005.

[CCR07a] J. Cabot, R. Clarisó, D. Riera. A tool for the formal verification of UML/OCL models using Constraint Programming. 2007. http://gres.uoc.edu/UMLtoCSP/.

[CCR07b] J. Cabot, R. Clarisó, D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 2007.

[CE08]   M. Clavel, M. Egea. The ITP/OCL tool. http://maude.sip.ucm.es/itp/ocl/, 2008.

[CED08]   M. Clavel, M. Egea, M. A. G. de Dios. Building an Efficient Component for OCL Evaluation. *ECEASST* 15, 2008.

[CRBG08] J. Carsí, I. Ramos, A. Boronat, A. Gómez. The MOMENT: MOdel ManageMENT Framework Project. http://moment.dsic.upv.es, 2008.

[CT07]   J. Cabot, E. Teniente. Transformation techniques for OCL constraints. *Science Computer Programming* 68(3):152–168, 2007.

[Dat06]   Database Systems Group. The UML Specification Environment (USE) tool. 2006. http://www.db.informatik.uni-bremen.de/projects/USE/.

[DCE08]   M. A. G. de Dios, M. Clavel, M. Egea. The Eye OCL Software (EOS). 2008. http://maude.sip.ucm.es/eos.

[Dem05]   B. Demuth. The OCL Portal. http://www-st.inf.tu-dresden.de/ocl/, 2005.

[DM08]   B. Dutertre, L. Moura. Yices: An SMT Solver. http://yices.csl.sri.com/, 2008.

[Hei06]   F. Heidenreich. OCL-Codegenerierung für Deklarative Sprachen. Master's thesis, University of Dresden, March 2006. http://dresden-ocl.sourceforge.net.

[Hus08]   K. Hussey. MDT-OCL. http://www.eclipse.org, 2008.

[Kon05]   A. Konermann. The Parser Subsystem of the Dresden OCL 2.0 Toolkit - Design and Implementation. http://dresden-ocl.sourceforge.net/publications.html, 2005.

[McC06]   W. McCune. Prover9. http://www.cs.unm.edu/~mccune/prover9/manual/June-2006C/, 2006.

[Sof07]   Software Technology Group. The OCL 2.0 Dresden Toolkit. http://sourceforge.net, 2007.