Noname manuscript No.
(will be inserted by the editor)

$\text{OCL}_{\textsf{FO}}$: First Order Expressive OCL Constraints for Efficient Integrity Checking

Enrico Franconi · Alessandro Mosca · Xavier Oriol · Guillem Rull · Ernest Teniente

Received: date / Accepted: date

Abstract OCL is the standard language for defining constraints in UML class diagrams. Unfortunately, as we show in this paper, full OCL is so expressive that it is not possible to check general OCL constraints efficiently. In particular, we show that checking general OCL constraints is not only not polynomial, but not even semidecidable. To overcome this situation, we identify $\text{OCL}_{\textsf{FO}}$, a fragment of OCL which is expressively equivalent to relational algebra (RA). By equivalent we mean that any $\text{OCL}_{\textsf{FO}}$ constraint can be checked through a RA query (which guarantees that $\text{OCL}_{\textsf{FO}}$ checking is efficient, i.e., polynomial), and any RA query encoding some constraint can be written as an $\text{OCL}_{\textsf{FO}}$ constraint (which guarantees expressiveness of $\text{OCL}_{\textsf{FO}}$). In this paper we define the syntax of $\text{OCL}_{\textsf{FO}}$, we concisely determine its semantics through set theory, and we proof its equivalence to RA. Additionally, we identify the core of this language, i.e. a minimal subset of $\text{OCL}_{\textsf{FO}}$ equivalent to RA.

Keywords OCL, relational algebra, translation, integrity checking

Enrico Franconi
Free University of Bozen-Bolzano, Italy
E-mail: franconi@inf.unibz.it

Alessando Mosca
SIRIS Academic, Spain
E-mail: a.mosca@sirisacademic.com

Xavier Oriol
Universitat Politècnica de Catalunya, Spain
E-mail: xoriol@essi.upc.edu

Guillem Rull
Universitat de Barcelona, Spain
E-mail: grull@essi.upc.edu

Ernest Teniente
Universitat Politècnica de Catalunya, Spain
E-mail: teniente@essi.upc.edu

1 Introduction

Since the definition of the Entity-Relationship (ER) language by Peter Chen in his seminal paper of 1976 [1], several new graphical modeling languages have been proposed so far by different researchers and institutions. Some prominent examples might be the Object Role Modeling language (ORM)[2], and the Unified Modeling language (UML) [3].

Using these languages, a software engineer can specify the conceptual schema of an information system. That is, the relevant concepts of the domain of interest, and how these concepts are related. Intuitively, the structural part of a conceptual schema (the one considered in this paper) consists of a class diagram, together with a set of textual constraints usually specified in a formal language such as OCL (Object Constraint Language) [4].

For instance, in Figure 1 we show a UML class diagram for some messaging application. The domain of such application consists of *users*, *conversation groups* (which can be divided into *pairs*, and *groups*) and *messages*. In this domain, users belong to conversation groups, and messages are sent by users to these groups.

To get a precise conceptual schema, this diagram has to be complemented with a set of *constraints*, i.e., conditions that the data it specifies should satisfy to be considered valid. Then, modeling languages provide themselves some graphical constructs that allow the definition of some frequent constraints. For instance, in our running example, we have stated the constraint that each message is sent to exactly one conversation group by means of two graphical UML cardinalities.

Moreover, and due to the limited expresiveness of graphical constraints, a *textual language* has to be considered also to express more sophisticated constraints.

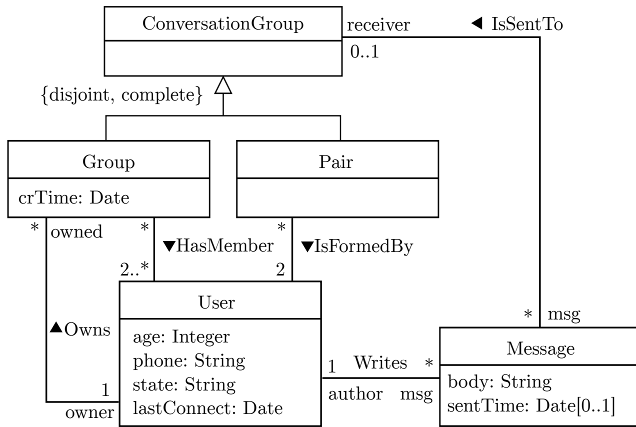

Fig. 1 UML class diagram

Indeed, constraints such as *Users only send messages to groups they belong to*, and *Messages of a group are sent after the group creation* cannot be graphically expressed in the previous UML diagram.

Currently, OCL (Object Constraint Language) [4] is probably the most popular notation to specify textual constraints and an ISO/IEC standard. Roughly, OCL permits defining constraints by means of building navigations from classes/associations, and applying some OCL operators to such navigations. For instance, the previously mentioned constraints can be specified in OCL as:

```
context Group inv MessagesAreFromGroup:
self.user->includesAll(self.msg.author)

context Group inv MessagesAreSentAfterCreation:
self.msg->forAll(m|m.sentTime > self.crTime)
```

Fig. 2 OCL constraints

However a natural question arising is to what extent is OCL expressive for defining constraints. That is, is OCL expressive enough for defining all the constraint we might require? or, on the contrary, is it even excessively expressive? In fact, the expressive power of a language determines the complexity to check or to analyze its expressions [5]. Therefore, we may want to avoid a language which is too expressive because it would penalize the time for its checking, but it may happen also that a restrictive language might be efficiently checked but useless.

This is an important question, which has no answer yet in the OCL literature. Probably, the closest study in this area is the one by Mandel and Cengarle in [6] but it was performed more than 15 years ago, and for an old version of OCL which did not include new constructs and capabilities that have been released since then. This is why we understand that a new and more careful analysis must be done. This is particularly important if we take into account that, in the OCL panel held in the OCL Workshop 2014, the OCL community discussed about whether OCL should include more operations (and which ones), or if OCL should just reorganize the current existing ones [7].

In this context, we show in this paper that full OCL is currently so expressive that it is able to encode non-decidable constraints (and even non-semidecidable ones). That is, we can write OCL constraints for which there is no algorithm able to check its satisfaction in finite time. It is important to hightlight that previous results only showed that OCL was undecidable for the problem of satisfiability (that is, whether it exists some data instance satisfying the constraint) [8], but we show that it is even undecidible with the satisfaction problem.

That means that OCL interpreters might not be able to assess whether an OCL constraint is satisfied by some data, even in the case that the data is indeed satisfying the constraint. Clearly, this result has a deep impact in the efficiency of OCL implementations like [9–11]. Indeed, our results not only imply undecidability, but also that decidable cases might be exponential. Thus, despite such implementations give support to most OCL expressions, many of them would only be feasible in toy examples rather than in real datasets. Our article brings light to what set of OCL operations can scale up for being checked with real big datasets.

In particular, in this article we identify a fragment of OCL which is expressive enough to write the most typically used textual constraints, but without loosing good computational properties for checking them. Our key idea is to look for the OCL fragment whose expressions can be checked through relational algebra (RA) queries in the sense that the constraint is violated iff the RA query returns a non-empty answer.

For instance, assume a translation from the UML class diagram in Figure 1 into a relational database such that each class gives rise to a different table with the same name, the same attributes plus an additional one corresponding to the OID (i.e. the object identifier, which is set for each tuple to a fresh value, different from the OID of any other tuple in the whole instance); and such that each association becomes also a table named as the association and with as many attributes as classes participate in the association (i.e. the OIDs of the participating classes).

Then, we have that the OCL constraints in Figure 2 can be checked by means of the following RA queries:

1. $\pi(\text{Group} \bowtie \text{IsSentTo} \bowtie \text{Writes}) \setminus \pi(\text{Group} \bowtie \text{HasMember})$
2. $\sigma_{\text{sentTime} \leq \text{crTime}}(\text{Group} \bowtie \text{IsSentTo} \bowtie \text{Message})$

Intuitively, the first query looks for the users which send messages to groups they do not belong to. In this way, one can check if the `MessagesAreFromGroup` constraint is satisfied by checking if this query is empty. Similarly, the second query looks for the messages whose sent time is previous to its group creation time. Note that, although not explicitly stated, the joins and the projections in the previous queries are performed through the OID attributes of the tables.

We name this fragment of OCL as $\text{OCL}_{\textsf{FO}}$[1]. We define the syntax of $\text{OCL}_{\textsf{FO}}$ with a formal grammar and determine its semantics by means of set theory. In this way, the language is fully described in an unambiguous and concise way so that such concise description may be easily understood and adopted by current practitioners. Note that, those approaches aimed at bringing formal semantics to the full OCL require much larger definitions, and even relying on third party languages [4,12,13].

Regarding its expressiveness, we show that $\text{OCL}_{\textsf{FO}}$ is not only *expressible* in relational algebra, but *equivalent* to relational algebra. That is, every $\text{OCL}_{\textsf{FO}}$ constraint can be checked by means of a relational algebra query, and every constraint that can be checked by means of a relational algebra query can be written in $\text{OCL}_{\textsf{FO}}$. This basic property ensures that $\text{OCL}_{\textsf{FO}}$ is as expressive as the main language of relational databases and guarantees that the complexity of checking the constraints is exactly the complexity of executing relational queries, that is, polynomial in data complexity (and in particular, AC^0). Moreover, it opens the door to reuse all the accumulated knowledge for efficient query answering in relational databases into efficiently checking of $\text{OCL}_{\textsf{FO}}$ constraints, as proposed by incremental approaches like the one in [14].

Finally, in order to make $\text{OCL}_{\textsf{FO}}$ an easy object of study, we also identify a *core* of the language that we call $\text{OCL}_{\textsf{CORE}}$. Indeed, $\text{OCL}_{\textsf{FO}}$ is targeted to include the fragment of OCL that can be encoded into relational algebra. On the one hand, this makes $\text{OCL}_{\textsf{FO}}$ a language containing most of the operators that an OCL practitioner might use. On the other hand, this makes $\text{OCL}_{\textsf{FO}}$ a difficult object of study since it inherits a lot of the OCL syntactic sugar. The *core* of the language is aimed at overcoming this situation since it is a minimal fragment of $\text{OCL}_{\textsf{FO}}$ (consisting only of 6 operations) able to express any constraint written in the whole $\text{OCL}_{\textsf{FO}}$.

The identification of this core of the language is also an important issue since it provides two significant contributions. First, it allows to easily state the relationship of any fragment of OCL with $\text{OCL}_{\textsf{FO}}$, by determining whether this fragment incorporates or not the six operations in $\text{OCL}_{\textsf{CORE}}$. Second, it entails that any implementation handling $\text{OCL}_{\textsf{CORE}}$ will also be able to deal with $\text{OCL}_{\textsf{FO}}$.

This paper improves and extends our previous work in [15], where the syntax and the semantics of $\text{OCL}_{\textsf{FO}}$ were initially identified, in the following terms:

- We prove that checking general OCL constraints is not decidable (and not even semidecidable).
- We revisit the syntax and semantics of $\text{OCL}_{\textsf{FO}}$ to include the major part of OCL that can be encoded into relational algebra.
- We provide the full formal proof of the equivalence between $\text{OCL}_{\textsf{FO}}$ and RA.
- We introduce the concept of $\text{OCL}_{\textsf{CORE}}$, a minimal subset of $\text{OCL}_{\textsf{FO}}$ with the same expressive power.

The paper is organized as follows. First, we define basic concepts in Section 2. Then, we show in Section 3 that checking general OCL constraints is not decidable. Afterwards, we present the language of $\text{OCL}_{\textsf{FO}}$ with its formal grammar and semantics in Section 4. In Sections 5 and 6 we prove that $\text{OCL}_{\textsf{FO}}$ constraints can be checked by relational algebra query emptiness, and viceversa. The $\text{OCL}_{\textsf{CORE}}$ is presented in Section 7. Finally, we review related work in Section 8, and discuss future work and conclusions in Section 9.

2 Basic conceps

UML class diagrams A *UML Class diagram* consists of a set of entity and relationship types. Entity types, also known as *classes*, have a name and a set of attributes; see for instance class User in our running example (Figure 1) with attributes phone, state and lastConnect. Relationship types, also called *associations*, are defined between pairs of entity types (we consider only binary relationship types); an example is Writes, defined between classes User and Message. Relationship types should not be confused with the so-called *is-a* relationship, which denote inheritance between classes; in our example, Pair and Group inherit from ConversationGroup, that is, they are subclasses of it, and it is superclass of them.

As we can see in Figure 1, UML class diagrams usually have graphical integrity constraints in the form of association cardinalities (e.g. Writes connects 1 author with 0 or more messages) or disjointness/completeness of entity type hierarchies (e.g. Group and Pair are disjoint, and each ConversationGroup is either a Group or a Pair). These are complemented with textual OCL constraints, which are the main focus of this paper.

[1] FO stands for First-Order, since relational algebra is, essentially, first-order logics.

An instance I of a UML class diagram S_{UML} is a finite set of objects (instances of classes) and links (instances of associations). An *object* is identified by an OID and has a value for each of the class' attributes. A *link* is a pair relating the OIDs of the connected objects. We say that I is *consistent* if it satisfies all constraints in S_{UML}.

OCL constraint An OCL constraint is a textual expression, attached to a UML class diagram, stating a condition that any UML instance for that schema should always satisfy. OCL constraints, as shown in Figure 2, are defined by means of a context, an invariant name, and a boolean expression.

The boolean expression of the OCL constraint should be satisfied for any given instance of the context class (referred as '*self*'). That is, given a UML instance, the OCL top boolean expression of the constraint should evaluate to true when interpreting *self* as any UML object of the given context class.

OCL offers several operators for building the OCL top boolean expression, where the most basic one is the *navigation* operation (referred as a '.'). Intuitively, a navigation from an object (set of objects) to some property, is an operation that, given the initial object (set of objects) collects the values/objects related to it according to such property. For instance, the constraint *MessagesAreFromGroup* from Figure 2 has the navigation *self.msg*. This navigation obtains the messages related to the group *self*. In addition, navigations can be naturally concatenated. Following with our example, *self.msg.author* obtains the users who are the authors of the messages related to *self*.

Given one or several OCL navigations, the OCL language provides several operators to check them such that it is possible to return a boolean value. For instance, in *MessagesAreFromGroup*, the OCL operator `includesAll` checks whether the contents of the navigation of *self.msg.author* is contained in the navigation *self.user*.

Relational schemas A relational schema is a set of relations $\{R_1, \ldots, R_n\}$, each with a particular signature $R(A_1, \ldots, A_m)$, where $A_1, \ldots, A_m$ are the attributes, and R is the relational symbol (with arity m). We usually identify a relation with its relational symbol.

An instance D of a relational schema S_{REL} is a set of tuples in the form of $R(a_1, \ldots, a_n)$, where R is a relational symbol from S_{REL} with arity n, and $a_1, \ldots, a_n$ are constants.

Relational views of UML class diagrams In this paper, we consider a particular class of relational schemas that are derived from UML class diagrams. We say that S_{REL} is the *relational view of UML class diagram* S_{UML} if it is a relational schema constructed by applying the following rules, starting from an empty set of relations:

- For each class C in S_{UML} with attributes $\mathsf{attr}_1, \ldots, \mathsf{attr}_n$, add n-ary relation C to S_{REL} with signature $C(oid, attr_1, \ldots, attr_n)$.
- For each association A in S_{UML} between any pair of classes C_1 and C_2, add binary relation A with signature $A(c_1, c_2)$ to S_{REL}.

Note that, assuming a UML class with n attributes, its relational translation is indeed a n+1-ary relation due to the addition of the OID. The concept of relational view is easy to reformulate at the level of instances if the OID of each tuple is set to a fresh value, different from the OID of any other tuple in the whole instance. The actual value of the OID is irrelevant, all that matters is its uniqueness with respect the OIDs of the other tuples. For the sake of simplicity, we assume single-valued, primitive-typed attributes.

More formally, let S_{REL} be the relational view of S_{UML}, and D and I instances of S_{REL} and S_{UML}, respectively. We say that D is the relational view of I if it is constructed from the empty instance, using the following rules:

- For each class $\mathsf{C} \in S_{\mathsf{UML}}$ and each object $c \in I$ of C, where $v_1, \ldots, v_n$ are the attribute values of c, add tuple $C(c, v_1, \ldots, v_n)$ to D.
- For each association $\mathsf{A} \in S_{\mathsf{UML}}$ and each link $a = (c_1, c_2) \in I$ instance of A, add tuple $A(c_1, c_2)$ to D.

A relational view D of a UML instance I is consistent if I is consistent.

Regarding graphical constraints, including generalization set constraints such as *incomplete* or *disjoint*, and their translation into the relational view, providing a specific translation for them would be redundant, as they can be expressed in OCL and translated into RA with the algorithms shown later in the paper. The unique exception are *IsA* hierarchy constraints, which can be encoded using relational foreign keys (subclass relations pointing to superclass relations).

Other UML aspects intended for software design rather than software specification (such as interfaces and directed associations) are outside of OCL_{FO}'s framework, and thus, outside of the scope of this article.

We use $\mathcal{R}_{\text{VIEWS}}$ to denote the class of relational views of UML class diagrams, i.e., $\mathcal{R}_{\text{VIEWS}} = \{S_{\mathsf{REL}} \mid \exists S_{\mathsf{UML}} : S_{\mathsf{REL}} \text{ is the relational view of } S_{\mathsf{UML}}\}$. Henceforth, whenever we say "relational schema", we mean a schema that belongs to $\mathcal{R}_{\text{VIEWS}}$.

Substitutions A substitution is a function that, when applied to an OCL_{FO} expression, replaces free variables

with constants. We use $E_{[s]}$ to denote the application of a substitution $s = \{var_1 \to c_1, var_2 \to c_2, \ldots\}$ to an expression E.

A substitution S_t obtained from some tuple t is a substitution that replaces variables with the values appearing in t. By default, S_t replaces a variable *var* with the value for the relational attribute called *var* appearing in t, i.e., $t[var]$. Conversely, we can also define a mapping $\mathcal{M}$ from OCL variables to relational attributes in t, so that, S_t replaces a variable *var* with $t[\mathcal{M}(var)]$.

Similarly, we say that a substitution S_t is obtained from some query q on some instance $\mathcal{I}$ when there exists some tuple $t \in q(\mathcal{I})$ from which S_t is obtained.

3 Undecidability of OCL Constraint Checking

It is well known that there is a tradeof between the expressiveness of a language and the computational complexity to *reason* with it. I.e., the greater its expressiveness, the more difficult is to *evaluate/analyze* its expressions.

Bearing this in mind, we are interested to know the computational complexity of checking a general OCL constraint in terms of *data complexity*. That is, how much difficult is to evaluate an OCL constraint regarding the size of a UML instantiation. Surprisingly, to our knowledge, although there are similar studies on other problems like OCL *maintenance* [16] or *reasoning* [17], this analysis has not been yet performed for integrity checking of OCL constraints.

In fact, it turns out that, unfortunately, current full OCL is so expressive that it is not decidable. That is, it is impossible to build an algorithm guaranteed to terminate and correctly assessing whether an OCL constraint is satisfied or not in an arbitrary UML instantiation. Things get even worse because we can also prove that OCL is not even semidecidable. Therefore, the algorithm is not ensured to terminate even in the case of UML instantiations that satisfy the constraint. Clearly, this entails a huge problem to OCL constraint evaluator techniques. Indeed, this result implies that checking techniques might hang when evaluating if some valid UML instantiation is valid, not to say that decidable cases might take an exponential amount of time.

We prove the previous results by means of reducing a non-decidable problem into OCL constraints checking. In particular, we reduce the problem of checking whether some word is accepted by a type-0 grammar [18] into OCL checking. Roughly speaking, a type-0 grammar is a set of replacing rules that transforms a finite string into another finite string. Thus, the word-acceptance problem in such grammar consists in, given some word, checking whether this word is generated by the set of production rules and some initial symbol. It is known that such problem is undecidable since, roughly speaking, a type-0 grammar can emulate a turing machine.

The reduction is based on exploiting OCL recursion, and OCL string operators. The idea is to build an OCL operation *produces* that recursively checks if a given word *target* is generated from an initial word *current* by means of a set of production rules. This is done by replacing the initial word into another one by using OCL String concatanation and substring operators. If the new word coincides with the *target*, the operation returns true, otherwise, there is a recursive call to check if transforming the new word we can generate the *target*.

The key of the undecidability relies on the finite, but unbounded, memory consumption required for such operation. Indeed, our OCL operation rewrites finite strings into new finite strings, but the size of the new strings cannot be bounded, which makes the recursion potentially infinite. This behavior mimics the one of the type-0 grammars (or turing machines). A Turing Machine might hang because of never reaching the final state consuming more and more tape (although it is a finite amount of tape). Equivalently, the type-0 grammar might derive, at runtime, non-terminal Strings of greater size each time, without never reaching the terminal word. This runtime derivation of non-terminal words of (potentially) increasing size is achieved in our setting through the recursive *produces* OCL operation, whose *current* input String might become longer and longer.

In the following, we formally state and proof such result.

Theorem 1 *Checking whether an OCL constraint is satisfied in an arbitrary UML instantiation is not decidable, and not even semidecidable.*

Proof First, we prove non-decidability. Then, we prove non-semidecidability.

To prove non-decidability, we make a reduction from a non-decidable problem to the problem of OCL integrity constraints checking. Thus, the undecidability of the former implies the undecidability of the latter. In particular, we make the reduction from the problem of word acceptance in a type-0 grammar. A type-0 grammar is a kind of grammar where all symbols (terminal, and non-terminal) might be substituted by means of the grammar rules. A formal definition of type-0 grammar might be found in [18], but, for our purposes, we only need to know that checking if a word is produced (aka accepted) by means of a type-0 grammar is undecidable.

Bearing this in mind, our proof strategy consists in building an OCL constraint stating that *a word is accepted, if and only if, it is produced by the grammar*. To do so, we define a UML class diagram encoding type-0 grammars on one side, and accepted/non accepted words on the other, so, we can define an OCL constraint imposing our intended constraint between the two.

In particular, our UML class diagram, shown in Figure 3, is capable of encoding all the production rules of an arbitrary type-0 grammar. A production rule is composed of a left and right hand sides, where each side is a sequence of symbols. These symbols might be classified into non-terminal, and terminal, where non-terminal symbols might be further classified into start symbols. Moreover, the UML class diagram also contains the notion of word, where words might be classified into accepted/non-accepted words. Note that, any type-0 grammar and accepted/non-accepted word can be described through this UML class diagram, i.e., any type-0 grammar and word can be written as an instance of this UML class diagram.

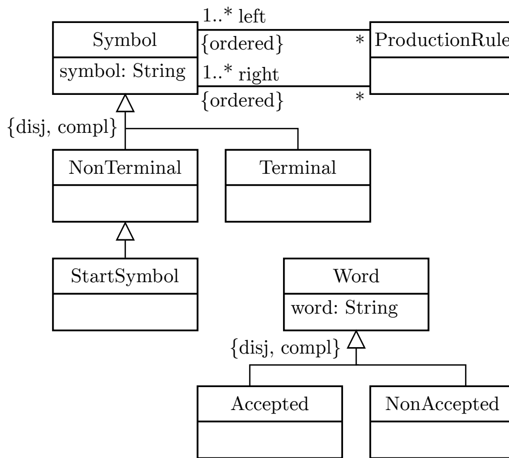

Fig. 3 UML class diagram for a 0 type grammar

Then, we add an OCL constraint assessing that instances of accepted words should be produced by the instances of production rules. We can build this OCL constraint by assessing that, given an accepted word, there should exists some production rule that produces it from the start symbol (either directly, or by means of subsequent production rule applications). This can be defined as follows:

```
context Accepted inv IsProducedWord:
ProductionRule.allInstances()->exists(p|
   StartSymbol.allInstances()->exists(s|
      p.produces(s.symbol,self.word))
```

where produces is an OCL operation that returns true if the first input string directly produces the second one, or if it produces some other word from which we can produce it. This operation can be defined in OCL as follows:

```
context ProductionRule def produces
(current: String, target: String): Boolean =
self.replacements(current)->exists(newWord|
   newWord = target or
   ProductionRule.allInstances()->exists(p|
      p.produces(newWord, target))
```

This definition makes use of the `replacements` operation, which returns the different words we can obtain by applying the production rule. `Replacements` can be defined recursively in OCL. Intuitively, we need to first compute the strings representing the left and right part of the production rule. Then, if the input word is empty, there is no replacement to apply and thus, we can return the empty set. Otherwise, we have to compute: 1) any replacement that can be performed from the first character, and 2) all the replacements that can performed from the second character and beyond. We can compute 1) by checking if the word's beginning matches the left part of the rule, and if so, add into the result the word corresponding to replace the beginning of the current word with the right part of the production rule. 2) can be computed recursively by means of skipping the first character, and then, iterate the given result to concatenate this first character at the beginning of each returned word. Formally:

```
context ProductionRule def replacements
(current: String): Set(String) =
let leftW: String = self.left
   ->iterate(s;l: String = "" | l+s.symbol) in
let rightW: String = self.right
   ->iterate(s;r: String = "" | r+s.symbol) in
if current = "" then Set{}
else if current.substring(1,leftW.size())=leftW
      then Set{rightW+
         current.substring(leftW.size(),
            current.size())}
      else Set{} endif
   ->union( self.replacements(
      current.substring(2,current.size())
      )->iterate(i; ac: Set{}|
      ac->including(current.substring(1,1)+i))
   )
endif
```

Thus, we can check whether some word is accepted by a type-0 grammar by instantiating the word and the grammar in the previous UML class diagram and checking the satisfaction of the given `IsProducedWord` OCL constraint. Since checking whether some word is accepted by a type-0 grammar is not decidable, check-

ing OCL constraints is not decidable either. This concludes the first part of the proof.

Now, to prove that checking OCL constraints is not even semidecidable, we reduce the problem of checking whether some word is rejected by some type-0 grammar, which is a well known non-semidecidable problem.

Let us consider a constraint specifying that non-accepted words are words that cannot be produced by the grammar. In OCL, this constraint can be stated by simply negating the previous one:

```
context NonAccepted inv IsNotProducedWord:
not ProductionRule.allInstances()->exists(p|
   StartSymbol.allInstances()->exists(s|
      p.produces(s.symbol,word.word))
```

We can check now whether some word is rejected by a type-0 grammar by instantiating the word and the grammar in the previous UML class diagram and checking the satisfaction of the given **IsNotProducedWord** OCL constraint. Therefore, checking OCL constraints is not even semidecidable. □

4 The OCL_{FO} fragment of OCL

We provide in this section the syntax and the semantics of OCL_{FO}. Since our goal is to use OCL_{FO} as a language for specifying constraints, we place special emphasis on OCL_{FO} boolean statements.

The syntax is defined through a formal grammar which limits the standard OCL boolean statements to those that can be computed through RA queries. Since, intuitively, relational algebra is known to be equivalent to (domain independent) first-order logic, such grammar leaves out the OCL higher-order operators like transitive closure, or (most) aggregation functions.

Semantics is defined by means of set theory. Roughly, OCL navigations are interpreted as sets, or single objects/values, and OCL boolean operators are interpreted as checks over them (e.g. the semantics of **includes** consists in checking whether some object/value belongs to a set, etc.). Therefore, this semantics does not distinguish among different collection types, as OCL does (e.g. it does not contemplate bags, nor ordered sets). However, this is not a drawback of OCL_{FO} but a necessary limitation to ensure equivalence with RA (since RA only supports sets). Furthermore, most of the OCL_{FO} considered operators regarding collections brings the same results when interpreting an OCL collection as a set. In fact, the unique ones that might differ are: **size**, **one**, and **isUnique** in case that its *collection source* included a navigation that might contain repeated objects.

After defining the syntax and the semantics of OCL_{FO}, we make a brief discussion about the OCL operations outside OCL_{FO} while distinguishing whether they could be effectively emulated in OCL_{FO}, or not. Then, we conclude this section analyzing the coverage of OCL_{FO} with two UML/OCL case studies.

4.1 OCL_{FO} Syntax

The grammar of OCL_{FO} is stated in Figure 4. Briefly, an OCL_{FO} constraint is an `OCL-Bool` statement written in some class context as shown in Figure 2. Such boolean statement might make use of *navigations*, i.e., `OCL-Set`, `OCL-Object`, or `OCL-Value` statements. Intuitively, the first kind of statements describe a set of objects, whereas the last two determine a single object/value, respectively. Such navigations are then used as the input of some OCL_{FO} operator to obtain the `OCL-Bool` statement that defines the constraint.

These OCL_{FO} statements are built over a *signature* consisting of a set of class names, role/attribute names (where some might be functional, i.e., with a maximum cardinality of 1), association class names, and constant names. Typically, this signature is provided by an associated UML class diagram.

For the sake of simplifying the language, we limit OCL_{FO} statements to evaluate to valid results. Thus, we require the expressions to apply the proper safety checks to avoid rising the OCL **invalid** value in runtime. For instance, if we have some object of type T_1, and we want to cast it to T_2, we might need to check that the object has also the type T_2 (unless T_1 is a subclass of T_2). Note that we can analyze in which cases are these safety checks necessary with a syntactic inspection of the OCL expression and the class diagram.

4.2 OCL_{FO} semantics

To define the semantics of OCL_{FO} we interpret the `OCL-Bool` statements as TRUE/FALSE values, the `OCL-Set` statements as sets of objects/values, and the `OCL-Object`/`OCL-Value` statements as a single object/value respectively.

We define first the semantics of OCL_{FO} without considering NULL values. That is, assuming that when interpreting an `OCL-Object` or `OCL-Value` expression, we always reach some defined object/value. We will include the treatment of *nulls* later on.

4.2.1 OCL_{FO} semantics without NULLS

The semantics of an OCL_{FO} statement is defined through the *interpretation* of its signature. Such inter-

```
OCL-Bool        ::=  OCL-Bool BoolOp OCL-Bool | not OCL-Bool |
                     OCL-Set ->includesAll(OCL-Set) | OCL-Set ->excludesAll(OCL-Set) |
                     OCL-Set ->includes(OCL-Single) | OCL-Set ->excludes(OCL-Single) |
                     OCL-Set ->forAll(VarList | OCL-Bool) |
                     OCL-Set ->exists(VarList | OCL-Bool) |
                     OCL-Set ->isEmpty() | OCL-Set ->notEmpty() |
                     OCL-Set ->size() CompOp Integer | ->one(Var | OCL-Bool) |
                     OCL-Set ->isUnique(attr) |
                     OCL-Object.oclIsKindOf(Class) | OCL-Object.oclIsTypeOf(Class) |
                     OCL-Object = null | OCL-Object <> null |
                     OCL-Navigation = OCL-Navigation | OCL-Navigation <> OCL-Navigation |
                     OCL-Value CompOp OCL-Value |
                     OCL-Object.bAttr | Var
OCL-Navigation  ::=  OCL-Set | OCL-Single
OCL-Set         ::=  OCL-Set ->union(OCL-Set) | OCL-Set ->intersection(OCL-Set) |
                     OCL-Set ->symmetricDifference(OCL-Set) | OCL-Set - OCL-Set |
                     OCL-Set ->select(Var | OCL-Bool) | OCL-Set ->reject(Var | OCL-Bool) |
                     OCL-Set ->selectByKind(Class) | OCL-Set ->selectByType(Class) |
                     OCL-Set.role [ [role] ] | OCL-Set.assoClass [ [role] ] |
                     OCL-Object.nfRole [ [role] ] | OCL-Object.nfAssoClass [ [role] ] |
                     OCL-Set.attr | OCL-Object.nfAttr |
                     Class.allInstances() | OCL-Single
OCL-Single      ::=  OCL-Object | OCL-Value
OCL-Object      ::=  OCL-Object.oclAsType(Class) |
                     OCL-Object.fRole | OCL-Object.fAssoClass |
                     Var | self
OCL-Value       ::=  Constant | Var
                     OCL-Object.fAttr |
                     OCL-Set->min() | OCL-Set->max() |
BoolOp          ::=  and | or | xor | implies
CompOp          ::=  < | <= | = | >= | > | <>
VarList         ::=  Var (,Var)*
Var             ::=  ⟨a variable name⟩
Class           ::=  ⟨a class name⟩
assoClass       ::=  ⟨an association class name⟩
fAssoClass      ::=  ⟨an association class name of a functional role⟩
nfAssoClass     ::=  ⟨an association class name of a non functional role⟩
role            ::=  ⟨a role name⟩
fRole           ::=  ⟨a functional role name⟩
nfRole          ::=  ⟨a non functional role name⟩
attr            ::=  ⟨an attribute name⟩
bAttr           ::=  ⟨a boolean attribute name⟩
fAttr           ::=  ⟨a functional attribute name⟩
nfAttr          ::=  ⟨a non functional attribute name⟩
Integer         ::=  ⟨an integer number⟩
Constant        ::=  ⟨a constant name⟩
```

Fig. 4 Syntax of $\mathrm{OCL}_{\mathsf{FO}}$

pretation represents a specific *database state* of the class diagram where the constraint is attached to. Namely, it indicates the classes each object is instance of, the relations between objects via associations, and the values objects have via their attributes. Since the interpretation of an association name determines the interpretation of its role names, instead of considering the interpretation of the roles, we assume a function $\mathsf{ass} : \mathtt{role} \mapsto \mathtt{Assoc}$, for retrieving the association name of some role name. In this way, the interpretation of some role r is obtained by taking the interpretation of its association $ass(r)$.

Formally, an *interpretation* is a pair $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$, where $\Delta^{\mathcal{I}}$ is a non-empty set of object identifiers and values referred as *the interpretation domain*, and $\cdot^{\mathcal{I}}$ is a function, referred as *interpretation function*, that maps each element in the signature of the $\mathrm{OCL}_{\mathsf{FO}}$ statements to $\Delta^{\mathcal{I}}$ tuples. In particular, class names are mapped to a set of domain elements, attribute names are mapped to a set of domain element pairs, k-ary association (class) names are mapped to a set of k-ary (k+1-ary) domain element tuples, and constant names are interpreted to domain elements with the same name (i.e., we follow the standard name assumption).

OCL-Bool^I	$\in$	{TRUE, FALSE}
(OCL-Bool$_1$ BoolOp OCL-Bool$_2$)^I	$\equiv$	OCL-Bool$_1{}^I$ BoolOp OCL-Bool$_2{}^I$
(**not** OCL-Bool)^I	$\equiv$	$\neg$OCL-Bool^I
(OCL-Set$_1$**->includesAll**(OCL-Set$_2$))^I	$\equiv$	OCL-Set$_1{}^I$ $\supseteq$ OCL-Set$_2{}^I$
(OCL-Set$_1$**->excludesAll**(OCL-Set$_2$))^I	$\equiv$	(OCL-Set$_1{}^I$ $\cap$ OCL-Set$_2{}^I$) $= \emptyset$
(OCL-Set**->includes**(OCL-Single))^I	$\equiv$	OCL-Single^I $\in$ OCL-Set^I
(OCL-Set**->excludes**(OCL-Single))^I	$\equiv$	OCL-Single^I $\notin$ OCL-Set^I
(OCL-Set**->forAll**(Var $\mid$ OCL-Bool))^I	$\equiv$	(**not** OCL-Bool)$^{I,\text{Var},\text{OCL-Set}}$ $= \emptyset$
(OCL-Set**->forAll**(VarList,Var $\mid$ OCL-Bool))^I	$\equiv$	(OCL-Set**->forAll**(Var $\mid$ OCL-Set**->forAll**(VarList $\mid$ OCL-Bool)))^I
(OCL-Set**->exists**(Var $\mid$ OCL-Bool))^I	$\equiv$	OCL-Bool$^{I,\text{Var},\text{OCL-Set}}$ $\neq \emptyset$
(OCL-Set**->exists**(VarList,Var $\mid$ OCL-Bool))^I	$\equiv$	(OCL-Set**->exists**(Var $\mid$ OCL-Set**->exists**(VarList $\mid$ OCL-Bool)))^I
(OCL-Set**->isEmpty()**)^I	$\equiv$	OCL-Set^I $= \emptyset$
(OCL-Set**->notEmpty()**)^I	$\equiv$	OCL-Set^I $\neq \emptyset$
(OCL-Set**->size()** CompOp n)^I	$\equiv$	$\lvert\lvert$OCL-Set^I$\rvert\rvert$ CompOp n
(OCL-Set**->one**(Var $\mid$ OCL-Bool))^I	$\equiv$	$\lvert\lvert$OCL-Bool$^{I,\text{Var},\text{OCL-Set}}$$\rvert\rvert = 1$
(OCL-Set**->isUnique**(attr))^I	$\equiv$	(OCL-Set**->forAll(v1,v2** $\mid$ **v1 <> v2 implies v1**.attr **<> v2**.attr))^I
(v**.oclIsKindOf**(Class))^I	$\equiv$	$v \in$ Class^I
(v**.oclIsTypeOf**(Class))^I	$\equiv$	$v \in$ Class^I $\setminus$ Subclasses(Class)^I
(OCL-Single **= null**)^I	$\equiv$	OCL-Single^I = NULL
(OCL-Single **<> null**)^I	$\equiv$	OCL-Single^I $\neq$ NULL
(OCL-Set$_1$ **=** OCL-Set$_2$)^I	$\equiv$	OCL-Set$_1{}^I$ = OCL-Set$_2{}^I$
(OCL-Set$_1$ **<>** OCL-Set$_2$)^I	$\equiv$	OCL-Set$_1{}^I$ $\neq$ OCL-Set$_2{}^I$
(OCL-Object$_1$ **=** OCL-Object$_2$)^I	$\equiv$	OCL-Object$_1{}^I$ = OCL-Object$_2{}^I$, or OCL-Object$_i$ = NULL
(OCL-Object$_1$ **<>** OCL-Object$_2$)^I	$\equiv$	OCL-Object$_1{}^I$ $\neq$ OCL-Object$_2{}^I$, or OCL-Object$_i$ = NULL
(OCL-Value$_1$ CompOp OCL-Value$_2$)^I	$\equiv$	OCL-Value$_1{}^I$ CompOp OCL-Value$_2{}^I$, or OCL-Value$_i{}^I$ = NULL
(OCL-Object$\bullet$bAttr)^I	$\equiv$	(OCL-Object$\bullet$bAttr)^I = TRUE, or (OCL-Object$\bullet$bAttr)^I = NULL
$(v)^I$	$\equiv$	v

OCL-Bool$^{I,\text{Var},\text{OCL-Set}}$	$=$	{$v \in$ OCL-Set^I $\mid$ (OCL-Bool$_{[\text{Var}/v]}$)^I = TRUE}

OCL-Set^I	$\subseteq$	Δ^I
(OCL-Set$_1$**->union**(OCL-Set$_2$))^I	$=$	OCL-Set$_1{}^I$ $\cup$ OCL-Set$_2{}^I$
(OCL-Set$_1$**->intersection**(OCL-Set$_2$))^I	$=$	OCL-Set$_1{}^I$ $\cap$ OCL-Set$_2{}^I$
(OCL-Set$_1$**->symmetricDifference**(OCL-Set$_2$))^I	$=$	OCL-Set$_1{}^I$ $\ominus$ OCL-Set$_2{}^I$
(OCL-Set$_1$ $-$ OCL-Set$_2$)^I	$=$	OCL-Set$_1{}^I$ $\setminus$ OCL-Set$_2{}^I$
(OCL-Set**->select**(Var $\mid$ OCL-Bool))^I	$=$	OCL-Bool$^{I,\text{Var},\text{OCL-Set}}$
(OCL-Set**->reject**(Var $\mid$ OCL-Bool))^I	$=$	OCL-Set^I $\setminus$ OCL-Bool$^{I,\text{Var},\text{OCL-Set}}$
(OCL-Set**->selectByKind**(Class))^I	$=$	OCL-Set^I $\cap$ Class^I
(OCL-Set**->selectByType**(Class))^I	$=$	OCL-Set^I $\cap$ Class^I $\setminus$ Subclasses(Class)^I
(OCL-Set$\bullet$role)^I	$=$	π_{role}(OCL-Set^I $\bowtie_{\mathsf{ns}}$ ass(role)^I)
(OCL-Set$\bullet$assoClass)^I	$=$	$\pi_{\text{assoClass}}$(OCL-Set^I $\bowtie_{\mathsf{ns}}$ (assoClass)^I)
(OCL-Object$\bullet$nfRole)^I	$=$	π_{nfRole}({OCL-Object^I} $\bowtie_{\mathsf{ns}}$ ass(nfRole)^I)
(OCL-Object$\bullet$nfAssoClass)^I	$=$	$\pi_{\text{nfAssoClass}}$({OCL-Object^I} $\bowtie_{\mathsf{ns}}$ (nfAssoClass)^I)
(OCL-Set$\bullet$attr)^I	$=$	π_{attr}(OCL-Set^I $\bowtie_{\mathsf{oid}}$(attr)^I)
(OCL-Object$\bullet$nfAttr)^I	$=$	π_{nfAttr}({OCL-Object^I} $\bowtie_{\mathsf{oid}}$(nfAttr)^I)
(Class.**allInstances()**)^I	$=$	Class^I
(OCL-Single)^I	$=$	{OCL-Single^I} if OCL-Single^I $\neq$ NULL, $\emptyset$ otherwise

OCL-Object^I	$\in$	$\Delta^I \cup$ {NULL}
(OCL-Object$\bullet$**oclAsType**(Class))^I	$=$	(OCL-Object)^I
(OCL-Object$\bullet$fRole)^I	$=$	π_{fRole}({OCL-Object}^{I} $\bowtie_{\mathsf{ns}}$ ass(fRole)^I), or NULL
(OCL-Object$\bullet$fAssoClass)^I	$=$	$\pi_{\text{fAssoClass}}$({OCL-Object^I} $\bowtie_{\mathsf{ns}}$ ass(fAssoClass)^I), or NULL
(v)^I	$=$	v

OCL-Value^I	$\in$	$\Delta^I \cup$ {NULL}
(v)^I	$=$	v
(OCL-Object$\bullet$fAttr)^I	$=$	π_{fAttr}({OCL-Object^I} $\bowtie_{\mathsf{oid}}$(fAttr)^I) , or NULL
(OCL-Set**->min()**)^I	$=$	(OCL-Set)^I $\setminus$ (π ($\sigma_{>}$(OCL-Set^I $\times$ OCL-Set^I)), or NULL
(OCL-Set**->max()**)^I	$=$	(OCL-Set)^I $\setminus$ (π ($\sigma_{<}$(OCL-Set^I $\times$ OCL-Set^I)), or NULL

Fig. 5 Semantics of OCL$_{\mathsf{FO}}$

For example, an *interpretation* $\mathcal{I}_0$ for the signature defined by our running UML class diagram example might be:

$\Delta^{\mathcal{I}_0} = \{\#user1, \#user2, \#group1, \#msg1,$
$1/1/2016, 12/12/2015, \text{'Happy new year!'}, \ldots\}$
$\texttt{User}^{\mathcal{I}_0} = \{\#user1, \#user2\}$
$\texttt{Group}^{\mathcal{I}_0} = \{\#group1\}$
$\texttt{Message}^{\mathcal{I}_0} = \{\#msg1\}$
$\texttt{crTime}^{\mathcal{I}_0} = \{< \#group1, 12/12/2015 >\}$
$\texttt{sentTime}^{\mathcal{I}_0} = \{< \#msg1, 1/1/2016 >\}$
$\texttt{isSentTo}^{\mathcal{I}_0} = \{< \#msg1, \#group1 >\}$

Given an interpretation $\mathcal{I}$, an OCL$_{\mathsf{FO}}$ statement is interpreted according to the recursive definition specified in Figure 5. Such definition is mainly provided in terms of set theory, together with some relational algebra operators (such as join $\bowtie$, project π, or select σ), to easily define the interpretation of OCL$_{\mathsf{FO}}$ navigations. Moreover, to define these navigations, we assume that *ns* is the role name of the navigation source which corresponds to the `[role]` expression of a navigation, or the opposite of some role in navigations through binary associations. Lastly, we use the expression *Subclasses(*`Class`*)*^{I} to refer to those objects belonging to a subclass of `Class`.

Thus, given an OCL$_{\mathsf{FO}}$ constraint ϕ of the form:

```
context R inv ConstraintName: OCL-Bool
```

We say that an interpretation $\mathcal{I}$ satisfies the constraint ϕ, and write $\mathcal{I} \models \phi$ if and only if it evaluates to TRUE for all the objects of its context class R. More formally:

$$\mathcal{I} \models \phi \textbf{ iff } \forall_{v \in R^I} \texttt{OCL-Bool}^I_{[\texttt{self}/v]} = \text{TRUE}$$

For example, $\mathcal{I}_0$ satisfies the OCL$_{\mathsf{FO}}$ constraint `MessagesAreSentAfterCreation` since for its unique group *#group1* it holds that:

$$\{v \in \pi_{msg}(\{\#group1\} \bowtie \texttt{isSentTo}^{I_0}) \mid \pi_{sentTime}(\{v\} \bowtie \texttt{sentTime}^{I_0}) < \pi_{crTime}(\{\#group1\} \bowtie \texttt{crTime}^{I_0})\} = \emptyset$$

In case $\mathcal{I}$ satisfies ϕ, we say that $\mathcal{I}$ is a *model* of ϕ, otherwise, we say that $\mathcal{I}$ violates ϕ. We naturally extend the notions of model, satisfaction, and violation to sets of OCL$_{\mathsf{FO}}$ constraints Φ. For instance, $\mathcal{I}$ satisfies a set of constraints Φ if and only if $\mathcal{I}$ satisfies each $\phi \in \Phi$.

4.2.2 OCL$_{\mathsf{FO}}$ semantics with nulls

Sometimes, the interpretation of some `OCL-Object` or `OCL-Value` results into no value. Indeed, consider that in our running example, some message *#msg1* has no value defined for the attribute *sentTime*. In this case, when navigating from the user *#msg1* to its *sentTime*, we obtain no value. More formally, we obtain $\emptyset$.

In such case, we define the `OCL-Object/OCL-Value` to be interpreted as a new value called NULL not present in Δ^I. In particular, we cast the $\emptyset$ to NULL (and viceversa) depending on the OCL expression it appears. Note that such interpretation corresponds to the one given in the OCL standard in [4].

Thus, when $\emptyset$ appears when interpreting some `OCL-Object/OCL-Value`, we automatically cast it to the new value NULL. Since the NULL value is not present in Δ^I, the NULL value does not join any value in the signature interpretation $\mathcal{I}$. That is, it does not join any value present in the interpretation of any association or attribute. This implies that, when navigating from a NULL value to obtain another object/value, we obtain again $\emptyset$, which is cast to NULL if this navigation is an `OCL-Object/OCL-Value`. This behavior perfectly emulates the standard OCL semantics proposal [4].

Finally, we need to extend the interpretation of the `OCL-Bool` to determine if they are evaluted to TRUE or FALSE when they use some `OCL-Object/OCL-Value` that evalutes to NULL. This differs from standard OCL since OCL considers that an `OCL-Bool` expression might return NULL or even INVALID. However, since we need OCL$_{\mathsf{FO}}$ to be a two-valued logic language (as it is first-order logics), we restrict `OCL-Bool` values to either TRUE or FALSE.

Thus, and following the criteria already used in [19], we consider that an `OCL-Bool` is true if some of its `OCL-Object/OCL-Value` subexpression are evaluated to NULL. The idea behind this interpretation is that an OCL$_{\mathsf{FO}}$ constraint is not violated unless the values that determine its satisfaction/violation are defined. For instance, the constraint `MessagesAreSentAfterCreation` would be satisfied in the previous example if *#msg1* had not *sentTime* defined yet, since then, its sent time would not be previous to the creation group time.

Note that this interpretation is just a default behavior to apply in case of finding a NULL value. However, note that it is possible to write a constraint that it is violated when some of its expressions evaluate to NULL by simply adding the boolean subexpression **and** `OCL-Single/OCL-Object` **`<> null`** to the OCL constraint.

4.3 OCL Operations not in OCL$_{\mathsf{FO}}$

There are two kinds of OCL operations that are not inside OCL$_{\mathsf{FO}}$. On one hand, we have operations that are outside the expressiveness of OCL$_{\mathsf{FO}}$, and on the

other, operations which we have intentionally left out for the sake of simplicity.

As we will prove in the next two sections, OCL$_{\mathsf{FO}}$ is expressively equivalent to relational algebra, and therefore, equivalent to (domain independent) first-order logic. Thus, it can be stated straightforwardly that all OCL operations which are not first-order (or not inside relational algebra) cannot be included in OCL$_{\mathsf{FO}}$.

Examples of operations that are not first-order are **`closure`**, or aggregation functions such as **`count`**. It is worth noting, however, that the aggregation function **`size`** can be used in OCL$_{\mathsf{FO}}$ to compare the size of some set with some fixed integer, but not with another set size neither with the value obtained from some attribute. Examples of operations that are outside because of relational algebra do not support them are basic type operations such as `+`, `-`, `*`, and `/`. In this last case, however, we argue that these operations could be integrated into relational algebra (e.g. in the selection operation), and thus, could be integrated into OCL$_{\mathsf{FO}}$.

For the sake of simplicity, we have left out operations like **`let ... in`**, **`if ... then ... else ... endif`**, **`including`** or **`excluding`**. This has been done for the sake of having a compact fragment of the language. We argue that these operations that we have left out are not frequently used when defining OCL constraints. Indeed, none of them appear, for instance, when defining constraints in the UML class diagrams of the DBLP case study [20] nor in the osCommerce UML/OCL case study [21], the case studies we have used to check the OCL$_{\mathsf{FO}}$ coverage.

In any case, we would like to highlight that, since OCL$_{\mathsf{FO}}$ captures relational algebra, it is as expressive and useful for defining constraints as, in essence, standard SQL assertions [22].

4.4 OCL$_{\mathsf{FO}}$ coverage evaluation through case studies

In this section, we discuss the expressiveness coverage of the OCL$_{\mathsf{FO}}$ fragment for defining constraints. By coverage, we mean the ratio of constraints that can be written in OCL$_{\mathsf{FO}}$ in comparison to full OCL, when considering conceptual schemas for real systems.

In the following, we first discuss the selected case studies (DBLP [20], and osCommerce [21]), and then, show and discuss the OCL$_{\mathsf{FO}}$ coverage.

4.4.1 The DBLP and osCommerce case studies

The DBLP and the osCommerce systems are two UML/OCL conceptual schemas describing real systems that have been used several times as case studies in conceptual modeling research.

The DBLP case study is a conceptual schema describing the DBLP system, the service that provides open bibliographic information on major computer science journals and proceedings. This UML/OCL conceptual schema consists of 17 classes, including concepts such as *Publication*, *ConferenceEdition*, *Journal*, etc.

The osCommerce case study is a conceptual schema describing the osCommerce platform, an open-source online store management program. This UML/OCL conceptual schema consists of around 40 classes, including concepts such as *Product*, *Order*, *PaymentMethod*, etc.

4.4.2 Evaluation of OCL$_{\mathsf{FO}}$

In the case of DBLP, 25 out of the 26 OCL constraints were inside the OCL$_{\mathsf{FO}}$ fragment. The unique constraint that remained outside was one speaking about *Sequences* which is a collection type we do not consider in OCL$_{\mathsf{FO}}$ since relational algebra works with unsorted collections.

In the case of osCommerce, 30 out of 33 OCL constraints were inside the OCL$_{\mathsf{FO}}$ fragment. Regarding the other 3, one was encodable in OCL$_{\mathsf{FO}}$ after rewriting (it was using an OCL *Tuple* construct that could be easily removed). The other two were constraints about transitive closure and collection size comparisons, which are outside first-order logics and thus, outside OCL$_{\mathsf{FO}}$.

Thus, from a total of 59 constraints extracted from real systems, we see that OCL$_{\mathsf{FO}}$ could encode directly 55, and could encode 56 after considering some rewriting. This represents a coverage of around 95%, which, in our opinion, makes the OCL$_{\mathsf{FO}}$ language useful in actual cases since its expressiveness allows to cover most of the constraints that are required in practice.

5 Checking OCL$_{\mathsf{FO}}$ Constraints by RA Queries

The main goal of this section is to show that any OCL$_{\mathsf{FO}}$ constraint can be checked by means of a RA query since given an OCL$_{\mathsf{FO}}$ constraint we can build a RA query that retrieves the objects that violate it. Therefore, the OCL$_{\mathsf{FO}}$ constraint is satisfied if and only if its corresponding query is empty.

This result entails that checking an OCL$_{\mathsf{FO}}$ constraint is at most as difficult as executing a RA query and, thus, checking an OCL$_{\mathsf{FO}}$ constraint can be solved in polynomial time with regarding to data complexity (and in particular, it belongs to the AC^0 complexity class). Moreover, this also enables reusing current

techniques developed in the community of relational databases for the treatment of OCL_{FO} constraints. For instance, incremental OCL_{FO} constraints checking can be solved by means of incremental query answering (e.g. following [14], or [23]), and repairing OCL_{FO} constraints can be solved by view updating techniques (as studied for instance in [16]).

The RA query is built from an OCL_{FO} constraint in two steps. First, we normalize the OCL_{FO} constraint into an equivalent one that uses a lower number of OCL_{FO} operators. Then, we translate the normalized OCL_{FO} constraint into a RA query that returns the objects that violate the constraint.

5.1 Normalizing OCL_{FO} Constraints

The grammar presented in Figure 4 admits very complex OCL expressions, using lots of different OCL operations. Therefore, defining directly a translation from pure OCL_{FO} constraints into RA queries becomes very cumbersome. To make things simpler, we initially translate each OCL_{FO} constraint into a *normalized* one, whose expression is defined only by means of a small number of operations.

We say that an OCL_{FO} constraint is normalized if it is defined only by means of the following operations: **and**, **not**, **forAll**, **=**, and **<** for `OCL-Bool` expressions; and **union**, **intersection**, **-**, **select** for `OCL-Set` expressions; although their expression can also contain the usual navigations through roles and attributes, and the **oclAsType** cast operation.

To normalize an OCL_{FO} constraint, we recursively apply the rewrittings defined in Figure 6. It is not difficult to verify that such rewritting preserve the original semantics of the constraint by means of directly inspecting the operation semantics defined in Figure 5.

In our running example, the normalized version of the constraint `MessagesAreFromGroup` is:

```
self.msg.author->forAll(a|
   not self.user->forAll(u|not a = u))
```

Note that the fragment of OCL obtained after normalization is only of internal use to facilitate the translation of OCL_{FO} to RA queries. So, we do not expect the designer to make use of this fragment when specifying integrity constraints since constraints in OCL_{FO} can be automatically normalized as stated before.

5.2 Drawing RA Queries from Normalized Constraints

To obtain the RA query of a normalized OCL_{FO} constraint, we first translate each OCL_{FO} navigation (i.e., `OCL-Set`, `OCL-Object` and `OCL-Value` expressions) into RA queries retrieving its corresponding set, object, or value. Then, the resulting RA query is obtained by translating the `OCL-Bool` expressions through the composition of the translation of its navigations.

More precisely, the crucial point for translating the navigations are the OCL_{FO} variables (i.e, the **self** variable and the iteration variables appearing in **forAll** and **select** expressions) since our goal is to build a RA query with one relational attribute for each OCL_{FO} variable alive in the OCL_{FO} navigation, together with one additional attribute containing the result of the navigation. For instance, when translating the normalized `MessagesAreFromGroup` constraint, the navigation **self**`.msg.author` is translated as a relational query with the attributes *self* and *result*.

Intuitively, when executing such queries over some interpretation $\mathcal{I}$, each retrieved tuple t represents a combination of values that the OCL_{FO} variables may take when evaluating the OCL_{FO} navigation with $\mathcal{I}$. For instance, if in the interpretation $\mathcal{I}$ we have that a group *#group1* has some message *#msg1* written by *#John*, then, the row <*#group1*, *#John*> appears in the query result that translates the navigation **self**`.msg.author`.

Then, the idea of composing those translations to obtain the translation of the whole `OCL-Bool` expression defining the constraint is to select those tuples that witness the violation of the boolean condition. That is, we want to select the row <*#group1*, *#John*> from the previous example in case that *#John* is not a member of *#group1*.

All translations are recursive and use the input variable q_c, the *context query*, which is the relational query that retrieves the values for the alive OCL_{FO} variables defined in the upper expression of the expression being translated. For instance, to translate **self**`.msg.author`, we need a context query q_c retrieving the values that the variable `self` might take, e.g. $q_c = \texttt{Group}$.

In the rest of this section, we first present the algorithms for translating each OCL_{FO} navigation, while discussing their intuition and providing a formal proof of their correctness. Afterwards, we show and prove how to use these algorithms to translate the whole `OCL-Bool` expression defining the OCL_{FO} constraint.

For the seek of simplicity, we omit some relational algebra low-level details such as relational algebra attribute renamings and some selection conditions since they can be easily understood from the context.

<table>
<tr><th colspan="3">OCL-Bool</th></tr>
<tr><td>OCL-Bool$_1$ or OCL-Bool$_2$</td><td>=</td><td>not (not OCL-Bool$_1$ and not OCL-Bool$_2$)</td></tr>
<tr><td>OCL-Bool$_1$ implies OCL-Bool$_2$</td><td>=</td><td>not OCL-Bool$_1$ or OCL-Bool$_2$</td></tr>
<tr><td>OCL-Bool$_1$ xor OCL-Bool$_2$</td><td>=</td><td>(OCL-Bool$_1$ or OCL-Bool$_2$) and (notOCL-Bool$_1$ or not OCL-Bool$_2$)</td></tr>
<tr><td>OCL-Set$_1$->includesAll(OCL-Set$_2$)</td><td>≡</td><td>OCL-Set$_2$->forAll(e$_2$| not OCL-Set$_1$->forAll(e$_1$| e$_1$<>e$_2$))</td></tr>
<tr><td>OCL-Set$_1$->excludesAll(OCL-Set$_2$)</td><td>≡</td><td>OCL-Set$_2$->forAll(e$_2$| OCL-Set$_1$->forAll(e$_1$| e$_1$<>e$_2$))</td></tr>
<tr><td>OCL-Set->includes(OCL-Single)</td><td>≡</td><td>not OCL-Set->forAll(e | e <> OCL-Single))</td></tr>
<tr><td>OCL-Set->excludes(OCL-Single)</td><td>≡</td><td>OCL-Set->forAll(e | e <> OCL-Single))</td></tr>
<tr><td>OCL-Set->exists(Var | OCL-Bool)</td><td>≡</td><td>not OCL-Set->forAll(Var | OCL-Bool)</td></tr>
<tr><td>OCL-Set->isEmpty()</td><td>≡</td><td>OCL-Set->forAll(e | 1 $\neq$ 1)</td></tr>
<tr><td>OCL-Set->notEmpty()</td><td>≡</td><td>not OCL-Set->forAll(e | 1 $\neq$ 1)</td></tr>
<tr><td>OCL-Set->size() < n</td><td>≡</td><td>OCL-Set->forAll(e$_1$, ..., e$_n$ | e$_1$=e$_2$ or e$_1$=e$_3$ or ... e$_{n-1}$=e$_n$)</td></tr>
<tr><td>OCL-Set->size() <= n</td><td>≡</td><td>OCL-Set->forAll(e$_1$, ..., e$_{n+1}$ | e$_1$=e$_2$ or e$_1$=e$_3$ or ... e$_n$=e$_{n+1}$)</td></tr>
<tr><td>OCL-Set->size() = n</td><td>≡</td><td>OCL-Set->size() <= n and not OCL-Set->size() < n-1</td></tr>
<tr><td>OCL-Set->one(Var | OCL-Bool)</td><td>≡</td><td>OCL-Set->select(Var | OCL-Bool)->size() = 1</td></tr>
<tr><td>OCL-Set->isUnique(attr)</td><td>≡</td><td>OCL-Set->forAll(v1,v2 | v1 <> v2 implies v1.attr <> v2.attr)</td></tr>
<tr><td>v.oclIsKindOf(Class)</td><td>≡</td><td>not Class->forAll(e | e <> v)</td></tr>
<tr><td>v.oclIsTypeOf(Class)</td><td>≡</td><td>not Class->forAll(e | e <> v) and Subclass->forAll(e | e <> v) ...</td></tr>
<tr><td>OCL-Single = null</td><td>≡</td><td>OCL-Single->forAll(e | 1 $\neq$ 1)</td></tr>
<tr><td>OCL-Single <> null</td><td>≡</td><td>not OCL-Single->forAll(e | 1 $\neq$ 1)</td></tr>
<tr><td>OCL-Set$_1$ = OCL-Set$_2$</td><td>≡</td><td>OCL-Set$_1$->includesAll(OCL-Set$_2$) and OCL-Set$_2$->includesAll(OCL-Set$_1$)</td></tr>
<tr><td>OCL-Set$_1$ <> OCL-Set$_2$</td><td>≡</td><td>not OCL-Set$_1$ = OCL-Set$_2$</td></tr>
<tr><th colspan="3">OCL-Set</th></tr>
<tr><td>OCL-Set$_1$->symmetricDifference(OCL-Set$_2$)</td><td>=</td><td>(OCL-Set$_1$ − OCL-Set$_2$)->union(OCL-Set$_2$ − OCL-Set$_1$)</td></tr>
<tr><td>OCL-Set->reject(Var | OCL-Bool)</td><td>=</td><td>OCL-Set->select(Var | not OCL-Bool)</td></tr>
<tr><td>OCL-Set->selectByKind(Class)</td><td>=</td><td>OCL-Set->select(e | e.oclIsKindOf(Class))</td></tr>
<tr><td>(OCL-Set->selectByType(Class))</td><td>=</td><td>OCL-Set->select(e | e.oclIsTypeOf(Class))</td></tr>
<tr><th colspan="3">OCL-Value</th></tr>
<tr><td>OCL-Set->min()</td><td>=</td><td>OCL-Set->select(min | OCL-Set->forAll(e | min <= e))</td></tr>
<tr><td>OCL-Set->max()</td><td>=</td><td>OCL-Set->select(max | OCL-Set->forAll(e | max >= e))</td></tr>
</table>

Fig. 6 OCL$_{\mathsf{FO}}$ normalization rewrittings

5.2.1 `OCL-Set` *Translation*

Algorithm 1 translates an `OCL-Set` into a relational query retrieving the same values/objects as the ones in the `OCL-Set`. Since the `OCL-Set` expression might have OCL free variables (such as **`self`**), we need the context query q_c to bring the different possible substitutions to apply to such variables. Thus, each tuple t in the result of the query has the form $< t_1, ..., t_n, v >$, where $t_1, ..., t_n$ represents a substitution for the OCL$_{\mathsf{FO}}$ variables appearing in the `OCL-Set` (which are taken from q_c), and v a value appearing in the `OCL-Set` according to such substitution for the free variables.

The idea behind the algorithm is to use the relational operation corresponding to each OCL$_{\mathsf{FO}}$ set operation. I.e., **`union`** is translated into $\cup$, role navigations as $\bowtie$, etc. The major difficult, however, relies on the translation of the **`select`** operation, which is translated using the translation of `OCL-Bool` expressions. In this case, the idea is to first, translate the `OCL-Set` source expression, and then, remove all those rows not satisfying the inner `OCL-Bool` expression.

As an example, consider the `OCL-Set` expression **`self.`**`msg`**`.`**`author` with a context query q_c=`Group` defining the values that the OCL$_{\mathsf{FO}}$ variable **`self`** might take. Such expressions is translated as:

$$\pi_{\text{group,author}}(\texttt{Group} \bowtie \texttt{IsSentTo} \bowtie \texttt{Writes})$$

Intuitively, the translation just translates the role navigations to RA joins, and then, projects the result to only retrieve the reachable *messages* for each *group*.

In the following we formally prove the correctness of this algorithm.

Theorem 2 *Let ϕ be an* `OCL-Set` *over a UML class diagram S_{UML}, $\overline{q}$ a relational algebra expression defined over the relational view of S_{UML}, and q_c a context query such that $\overline{q}$ = raTranslation(ϕ, q_c) (Algorithm 1). Then,*

$$v \in \phi^{I}_{[S_t]} \textbf{ iff } < t[0], ..., t[n], v > \ \in \overline{q}(I)$$

for any value v, any interpretation $\mathcal{I}$, and any substitution S_t obtained from $q_c(I)$.

Proof The proof is based on structural induction over the OCL$_{\mathsf{FO}}$ grammar. In the following we bring the proof for the base case and one inductive case. The rest of cases follows analogously. Consider the base case:

$$\phi = \texttt{R.}\textbf{\texttt{allInstances()}}$$

Algorithm 1 raTranslation(OCL-Set, q_c)

```
if OCL-Set = OCL-Set1->union(OCL-Set2) then
    q1 := raTranslation(OCL-Set1, qc)
    q2 := raTranslation(OCL-Set2, qc)
    return q1 ∪ q2
else if OCL-Set = OCL-Set1->intersection(OCL-Set2) then
    q1 := raTranslation(OCL-Set1, qc)
    q2 := raTranslation(OCL-Set2, qc)
    return q1 ⋈ q2
else if OCL-Set = OCL-Set1− OCL-Set2 then
    q1 := raTranslation(OCL-Set1, qc)
    q2 := raTranslation(OCL-Set2, qc)
    return q1 \ q2
else if OCL-Set = OCL-Set1->select(Var|OCL-Bool) then
    qs := raTranslation(OCL-Set1, qc)
    qr := raTranslation(OCL-Bool, qs)
    return qs \ qr
else if OCL-Set = OCL-Set1.role then
    q1 := raTranslation(OCL-Set1, qc)
    return π(q1 ⋈ ass(role))
else if OCL-Set = OCL-Set1.assoClass then
    q1 := raTranslation(OCL-Set1, qc)
    return π(q1 ⋈ assoClass)
else if OCL-Set = OCL-Object1.nfRole then
    q1 := raTranslation(OCL-Object1, qc)
    return π(q1 ⋈ ass(nfRole))
else if OCL-Set = OCL-Object1.nfAssoClass then
    q1 := raTranslation(OCL-Object1, qc)
    return π(q1 ⋈ nfAssoClass)
else if OCL-Set = OCL-Set1.attr then
    q1 := raTranslation(OCL-Set1, qc)
    return π(q1 ⋈ attr)
else if OCL-Set = OCL-Object1.nfAttr then
    q1 := raTranslation(OCL-Object1, qc)
    return π(q1 ⋈ nfAttr)
else if OCL-Set = R.allInstances() then
    return qc × R
else if OCL-Set = OCL-Object1 then
    return raTranslation(OCL-Object1, qc)
end if
```

According to Algorithm 1,

$$\overline{q} = \text{raTranslation}(\phi, q_c) = q_c \times R$$

Clearly, according to the semantics of OCL_{FO}, $v \in$ $\texttt{R.allInstances()}^I$ iff $v \in \texttt{R}^I$. Moreover, it is guaranteed that S_t is obtained from $q_c(I)$. Hence, $v \in \texttt{R.allInstances()}^I_{[S_t]}$ iff $< t[0], ..., t[n], v > \ \in q_c(I) \times \texttt{R}^I$.

Consider the inductive case:

$$\phi = \texttt{OCL-Set.select(s | OCL-Bool)}$$

According to Algorithm 1,

$$\overline{q} = \text{raTranslation}(\phi, q_c) = q_s \setminus q_r$$

where

$$q_s = \text{raTranslation}(\texttt{OCL-Set}, q_c)$$
$$q_r = \text{raTranslation}(\texttt{OCL-Bool}, q_s)$$

According to the semantics, we have that $v \in \texttt{OCL-Set.select(s | OCL-Bool)}^I_{[S_t]}$ if and only if $v \in \texttt{OCL-Bool}^{I,\texttt{s},\texttt{OCL-Set}}_{[S_t]}$. This is the case if and only if $v \in \texttt{OCL-Set}^I_{[S_t]}$ and $\texttt{OCL-Bool}^I_{[S_t \cup \{\texttt{s}/v\}]} =$ TRUE. Equivalently, this is the case iff $v \in \texttt{OCL-Set}^I_{[S_t]}$ and not $\texttt{OCL-Bool}^I_{[S_t \cup \{\texttt{s}/v\}]} =$ FALSE. Now, by induction hypothesis, we have that $v \in \texttt{OCL-Set}^I_{[S_t]}$ iff $< t[0], ..., t[n], v > \ \in q_s(I)$, and $\texttt{OCL-Bool}^I_{[S_t \cup \{\texttt{s}/v\}]} =$ FALSE iff $< t[0], ..., t[n], v > \ \in q_r(I)$. Hence, $v \in \texttt{OCL-Set.select(s | OCL-Bool)}^I_{[S_t]}$ iff $< t[0], ..., t[n], v > \ \in \overline{q}(I)$. □

5.2.2 `OCL-Object` *and* `OCL-Value` *Translation*

Algorithm 2 defines the translation of an `OCL-Object` or `OCL-Value` expression into a relational query that, intuitively, returns the object/value referred by the expression. Similarly as before, each tuple $< t_1, ..., t_n, v >$ returned by the query represents an evaluation that the OCL_{FO} variables appearing in the expression might take (values $t_1, ..., t_n$ taken from a context query q_c), together with the value retrieved by the `OCL-Object`/`OCL-Value` expression according to that evaluation (v).

Again, the idea behind the translation is to use the relational algebra operator that corresponds to those defined in the OCL_{FO} semantics. E.g, role navigations are translated by means of the same join we have defined in the OCL_{FO} semantics.

For instance, consider the `OCL-Single` expression `m.sentTime` with a context query $q_c = \pi_{\texttt{msg}}(\texttt{Group} \bowtie \texttt{IsSentTo})$ defining the values for the OCL_{FO} variable `m`. Such expression is translated as:

$$(\pi_{\texttt{msg}}(\texttt{Group} \bowtie \texttt{IsSentTo})) \bowtie \texttt{SentTime}$$

Intuitively, the translation converts the attribute navigation as a new join to retrieve the *sentTime* attribute for each value that `m` might take.

In the following, we formally proof the correctness of Algorithm 2.

Algorithm 2 raTranslation(OCL-Single, q_c)

```
if OCL-Single = Constant then
    return qc × {Constant}
else if OCL-Single = Variable then
    return qc
else if OCL-Single = OCL-Object1.fAttr then
    q1 := raTranslation(OCL-Object1, qc)
    return π (q1 ⋈ fAttr)
else if OCL-Single = OCL-Object1.oclAsType(Class) then
    return raTranslation(OCL-Object1, qc)
else if OCL-Single = OCL-Object1.fRole then
    q1 := raTranslation(OCL-Object1, qc)
    return π(q1 ⋈ ass(fRole))
else if OCL-Set = OCL-Object1.fAssoClass then
    q1 := raTranslation(OCL-Object1, qc)
    return π(q1 ⋈ fAssoClass)
end if
```

Theorem 3 *Let ϕ be an* `OCL-Single` *over a UML class diagram S_{UML}, $\overline{q}$ a relational algebra expression defined*

over the relational view of $S_{\textsf{UML}}$, and q_c a context query such that $\overline{q} = raTranslation(\phi, q_c)$ (Algorithm 2). Then, for any interpretation $\mathcal{I}$, and any substitution S_t obtained from $q_c(I)$, we have:

$$\text{NULL} = \phi^{I}_{[S_t]} \textbf{ iff } \neg\exists v. < t[0], ..., t[n], v > \;\in \overline{q}(I)$$

and, for any value v different from NULL*:*

$$v = \phi^{I}_{[S_t]} \textbf{ iff } < t[0], ..., t[n], v > \;\in \overline{q}(I)$$

Proof The proof is based on structural induction over the $\text{OCL}_{\textsf{FO}}$ grammar. In the following we bring the proof for one base case and one inductive case. The rest of cases follows analogously. Consider the base case:

$$\phi = \texttt{self}$$

According to Algorithm 2,

$$\overline{q} = \text{raTranslation}(\phi, q_c) = q_c \quad (1)$$

According to the semantics, we have that $v = \texttt{self}^{I}_{[S_t]}$ iff $\texttt{self}_{[S_t]} = v$. This is the case if and only if we have $< t[0], ..., v, ..., t[n] > \;\in q_c$.

Consider the inductive case:

$$\phi = \texttt{OCL-Object.fAttr}$$

According to Algorithm 2,

$$\overline{q} = \pi(q_1 \bowtie \texttt{fAttr})$$

where

$$q_1 = \text{raTranslation}(\texttt{OCL-Object}, q_c)$$

According to the semantics, we have that $v = (\texttt{OCL-Object.fAttr})^{I}_{[S_t]}$ iff $v = \pi\ (\texttt{OCL-Object}^{I}_{[S_t]} \bowtie \texttt{fAttr}^{I})$. This is the case if and only if there exists some value v' in $\texttt{OCL-Object}^{I}_{[S_t]}$ whose join with $\texttt{fAttr}^{I}$ retrieves v. By induction hypothesis we know that $< t[0], ..., t[n], v' > \;\in q_1(I)$ iff $v' = \texttt{OCL-Object}^{I}_{[S_t]}$. Thus, $v = (\texttt{OCL-Object.fAttr})^{I}_{[S_t]}$ iff $< t[0], ..., t[n], v > \;\in \overline{q}(I)$. □

5.2.3 OCL-Bool Translation Algorithm

In Algorithm 3 we show how to make use of the previous translations to obtain the values that cause the violation of the `OCL-Bool` condition. The output of this algorithm is a query that returns the evaluation of the $\text{OCL}_{\textsf{FO}}$ variables alive in `OCL-Bool` that make the expression evaluate to FALSE.

The intuition behind the translation is to use the relational algebra selection σ to select those values that contradict the `OCL-Bool` expression.

For instance, consider the `OCL-Bool` expression **`self.user->forAll(u|not a = u)`** with the context query $q_c = \pi_{\texttt{group,author}}(\texttt{Group} \bowtie \texttt{IsSentTo} \bowtie \texttt{Writes})$ defining the values for the $\text{OCL}_{\textsf{FO}}$ variable `a` depending on the value given to **`self`**. Such expression would be translated as:

$$\sigma_{\texttt{author=user}}(q_c \bowtie (q_c \bowtie \texttt{HasMember}))$$

Intuitively, q_c retrieves the values that `a` might take for every value of **`self`** (that is, all the authors of messages sent to some group **`self`**), then, $(q_c \bowtie \texttt{IsMemberOf})$ retrieves the values that `u` might take for every value of **`self`** (that is, all the users of some group **`self`**). Then, the join of both expressions retrieves the values that `a` and `u` might take for the same value of **`self`** (that is, all the authors and members of some group **`self`**). Finally, the selection picks those tuples in which the value for `a` is equal to the value for `u`.

Note that, to translate the other normalized $\text{OCL}_{\textsf{FO}}$ boolean operations such as **`and`** and **`not`** we just need to compose the previous translation pattern. That is, **`and`** is translated by unifying the set of rows that causes the violation of the first condition, with those causing the violation of the second one; and **`not`** is translated by computing those rows violating the inner expression (in other words, those rows satisfying its negation), and removing them from the context query (so we have those rows violating the negated statement).

In the following, we formally proof the correctness of Algorithm 3.

Algorithm 3 raTranslation(`OCL-Bool`, q_c)

if `OCL-Bool` $=$ $\texttt{OCL-Bool}_1$ **`and`** $\texttt{OCL-Bool}_2$ **then**
    q_1 = raTranslation($\texttt{OCL-Bool}_1$, q_c)
    q_2 = raTranslation($\texttt{OCL-Bool}_2$, q_c)
    return $q_1 \cup q_2$
else if `OCL-Bool` $=$ **`not`** $\texttt{OCL-Bool}_1$ **then**
    return $q_c \setminus$ raTranslation($\texttt{OCL-Bool}_1$, q_c)
else if `OCL-Bool` $=$ `OCL-Set`**`->forAll(`**`Var|`$\texttt{OCL-Bool}_1$**`)`** **then**
    q_s := raTranslation(`OCL-Set`, q_c)
    q_b := raTranslation($\texttt{OCL-Bool}_1$, q_s)
    return $\pi\ q_b$
else if `OCL-Bool` $=$ $\texttt{OCL-Single}_1$ `CompOp` $\texttt{OCL-Single}_2$ **then**
    q_1 = raTranslation($\texttt{OCL-Single}_1$, q_c)
    q_2 = raTranslation($\texttt{OCL-Single}_2$, q_c)
    return $\pi\sigma\ q_1 \bowtie q_2$
else if `OCL-Bool` $=$ $\texttt{OCL-Value}_1$ **then**
    q_1 := raTranslation($\texttt{OCL-Value}_1$, q_c)
    return $\pi\sigma\ q_1$
end if

Theorem 4 *Let ϕ be an* `OCL-Bool` *over a UML class diagram $S_{\textsf{UML}}$, $\overline{q}$ a relational algebra expression defined over the relational view of $S_{\textsf{UML}}$, and q_c a context query such that $\overline{q} = raTranslation(\phi, q_c)$ (Algorithm 3). Then,*

for any interpretation $\mathcal{I}$, and any substitution S_t obtained from $q_c(I)$, we have:

$$\phi^I_{[S_t]} = \text{FALSE} \textbf{ iff } t \in \overline{q}(I)$$

Proof The proof is based on structural induction over the OCL_{FO} grammar. In the following we bring the proof for the base case and one inductive case. The rest of cases follows analogously. For the base case, we consider the expression:

$$\phi = \texttt{OCL-Single}_1 \ \texttt{CompOp} \ \texttt{OCL-Single}_2$$

According to Algorithm 3,

$$\overline{q} = \pi\sigma(q_1 \bowtie q_2)$$

where

$$q_1 = \text{raTranslation}(\texttt{OCL-Single}_1, q_c)$$
$$q_2 = \text{raTranslation}(\texttt{OCL-Single}_2, q_c)$$

According to the semantics, $\phi^I_{[S_t]} = \text{FALSE}$ iff the values $v_1 = \texttt{OCL-Single1}^I_{[S_t]}$, and $v_2 = \texttt{OCL-Single}^I_{[S_t]}$ do not satisfy the comparison operator CompOp. By induction hypothesis we have that $< t[0], ..., t[n], v_1 > \ \in q_1(I)$, and $< t[0], ..., t[n], v_2 > \ \in q_2(I)$. Thus, we can obtain the values v_1 and v_2 by joining q_1 and q_2, and select the row $< t[0], ..., t[n] >$ iff v_1 and v_2 do not satisfy the corresponding CompOp. Thus, $\phi^I_{[S_t]} = \text{FALSE}$ iff $t \in \overline{q}(I)$.

For the inductive case, we consider:

$$\phi = \texttt{OCL-Set->}\textbf{\texttt{forAll}}\texttt{(Var|OCL-Bool)}$$

According to Algorithm 3,

$$\overline{q} = \pi q_b$$

where

$$q_b = \text{raTranslation}(\texttt{OCL-Bool}, q_s)$$
$$q_s = \text{raTranslation}(\texttt{OCL-Set}, q_c)$$

According to the semantics, we have that $\texttt{OCL-Set->}\textbf{\texttt{forAll}}\texttt{(Var|OCL-Bool)}^I_{[S_t]} = \text{FALSE}$ iff $(\textbf{\texttt{not}} \ \texttt{OCL-Bool})^{I,\texttt{Var},\texttt{OCL-Set}}_{[S_t]} \neq \emptyset$. This is the case iff $\exists v.v \in \texttt{OCL-Set}^I_{[S_t]}$ and $\texttt{OCL-Bool}^I_{[S_t \cup \{\texttt{Var}/v\}]} = \text{FALSE}$. By induction hypothesis, we have that, for any value v, $v \in \texttt{OCL-Set}^I_{[S_t]}$ if and only if $< t[0], ..., t[n], v > \ \in q_s(I)$. Moreover, by induction again we know that $v \in \texttt{OCL-Set}^I_{[S_t]}$ and $\texttt{OCL-Bool}^I_{[S_t \cup \{\texttt{Var}/v\}]} = \text{FALSE}$ if and only if $< t[0], ..., t[n], v > \ \in q_b(I)$. Thus, $\texttt{OCL-Set->}\textbf{\texttt{forAll}}\texttt{(Var|OCL-Bool)}^I_{[S_t]} = \text{FALSE}$ if and only if $t \in \overline{q}(I)$. □

5.2.4 Translating an OCL_{FO} Constraint

To translate an OCL_{FO} constraint, it is enough to invoke Algorithm 3 with the body of the constraint as the `OCL-Bool` parameter and the context class in which the constraint is defined as the context query q_c, so that, the variable **`self`** is going to be evaluated to all the objects of the given context class.

For instance, consider the constraint of our running example `MessagesAreFromGroup`. This constraint would be translated into:

$$\pi(\texttt{Group} \bowtie \texttt{IsSentTo} \bowtie \texttt{Writes}) \setminus$$
$$\pi\sigma((\texttt{Group} \bowtie \texttt{IsSentTo} \bowtie \texttt{Writes}) \bowtie$$
$$(\texttt{Group} \bowtie \texttt{IsSentTo} \bowtie \texttt{Writes} \bowtie \texttt{HasMember}))$$

Intuitively, the query picks up all *users* who have written in some group (first line of the translation), and it takes out all those *users* who are indeed members of such group (second and third line of the translation). Thus, note that the constraint is satisfied if and only if the previous query is empty.

In the following we formally proof the correctness of the translation.

Theorem 5 *Let ϕ be an OCL_{FO} constraint over a UML class diagram S_{UML}, defined on the context class R, and $\overline{q}$ a relational algebra expression, defined over the relational view of S_{UML}, such that $\overline{q}$ = raTranslation(ϕ, R) (Algorithm 3). Then, for any interpretation $\mathcal{I}$, we have:*

$$I \models \phi \textbf{ iff } \overline{q}(I) = \emptyset$$

Proof Directly from Theorem 4 we have that $\overline{q}(I)$ retrieves those values for the OCL_{FO} variable **`self`** from R^I s.t. that makes the `OCL-Bool` in ϕ evaluate to FALSE. Thus, $I \models \phi$ iff $\overline{q}(I) = \emptyset$. □

It is worth noting that the translation algorithm is linear w.r.t. the number of OCL operators and OCL variables of the constraint. Indeed, every OCL operator is translated for a fixed number of RA operations (1 in most cases, 3 in the worst case), and every OCL variable reference is translated as a RA query whose size is limited by the size of the OCL source it ranges from. Thus, the final size of the translation of an OCL constraint defined by o operations and v variables can be upper-bounded by $3o + 3ov$. Since every OCL operator is traversed only once in the algorithm, this result is computed in the same upper-bound amount of time. In any case, this computation is absolutely independent from the data, which means that data-complexity, the time that takes to analyze an OCL_{FO} constraint w.r.t. the amount of the data stored in the information system, is not affected by it. Intuitively, this is because this

translation can be seen as a compilation step to apply only once at the beginning, and reuse forever.

6 RA queries to OCL$_{\mathsf{FO}}$ Constraints

Now, we show that any constraint that can be checked by means of a relational algebra query can be encoded as an OCL$_{\mathsf{FO}}$ constraint. That is, for any relational query q, we can build an OCL$_{\mathsf{FO}}$ constraint ϕ such that, for any interpretation $\mathcal{I}$, we have that $q(\mathcal{I}) = \emptyset$ iff $I \models \phi$.

This result implies that the language of OCL$_{\mathsf{FO}}$ is as expressive for defining constraints as relational algebra. Taking into account that in Theorem 5 we showed that any OCL$_{\mathsf{FO}}$ constraint can be checked by means of a relational query, we may conclude that OCL$_{\mathsf{FO}}$ is exactly as expressive for defining constraints as relational algebra. Thus, checking OCL$_{\mathsf{FO}}$ constraints is as difficult as executing a relational query, that is polynomial with regarding to data complexity (and AC^0 in particular).

We define the *oclTranslation* from a RA query to an OCL$_{\mathsf{FO}}$ constraint in Algorithm 4. This algorithm receives three input parameters: a context query q_c, an OCL$_{\mathsf{FO}}$ boolean statement `OCL-Bool`, and a mapping $\mathcal{M}$ that makes explicit which attributes from q_c are mapped to which OCL$_{\mathsf{FO}}$ variables from `OCL-Bool`. Then, the idea is that the algorithm returns a new OCL$_{\mathsf{FO}}$ boolean statement ϕ such that, for any given interpretation $\mathcal{I}$, $\mathcal{I} \models \phi$ iff $\texttt{OCL-Bool}^{\mathcal{I}}_{[S_t]} = \text{TRUE}$ for any given substitution S_t obtained from $q_c(\mathcal{I})$.

Then, we can then obtain the OCL$_{\mathsf{FO}}$ constraint corresponding to a relational query q by invoking *oclTranslation(q,* **`false`**,$\emptyset$*)*. Indeed, the unique way that for all substitutions S_t obtained from q we can have $\texttt{false}^{\mathcal{I}}_{[St]} = \text{TRUE}$ is that $q(\mathcal{I}) = \emptyset$.

Intuitively, the algorithm works by recursively removing relational operators from the input query q and placing them in the OCL-boolean expression given as a parameter. For instance, if we invoke:

oclTranslation(`R \ S`*,* **`false`***,* $\emptyset$*)*

We first recursively translate:

oclTranslation(`S`, `r <> s`*,* $\{\texttt{s} \rightarrow S\}$*)*

to obtain an OCL$_{\mathsf{FO}}$ boolean expression that characterizes those values of some variable `r` that are different to any element in `S`. In particular, we obtain the new OCL-boolean ϕ:

```
S.allInstances()->forAll(s|r <> s)
```

Then, we recursively translate:

oclTranslation(`R`, ϕ **`implies false`**, $\{r \rightarrow R\}$*)*

Algorithm 4 oclTranslation(q_c, `OCL-Bool`, $\mathcal{M}$)

if $q_c = q_1 \cup q_2$ **then**
  `OCL-Bool`$_1$ = oclTranslation(q_1, `OCL-Bool`, $\mathcal{M}$)
  `OCL-Bool`$_2$ = oclTranslation(q_2, `OCL-Bool`, $\mathcal{M}$)
  return `OCL-Bool`$_1$ **`and`** `OCL-Bool`$_2$
else if $q_c = \pi_a q_1$ **then**
  return oclTranslation(q_1, `OCL-Bool`, $\mathcal{M}$)
else if $q_c = \sigma_{a\omega b}\ q_1$ **then**
  $\mathcal{M}'$:= getCompleteMap($\mathcal{M}$, q_1)
  `OCL-Bool'` := $\mathcal{M}'$.getVar(`q.a`) ω $\mathcal{M}'$.getVar(`q.b`) **`implies`** `OCL-Bool`
  return oclTranslation(q_1, `OCL-Bool'`, $\mathcal{M}'$)
else if $q_c = q_1 \times q_2$ **then**
  `OCL-Bool`$_2$:= oclTranslation(q_2, `OCL-Bool`, $\mathcal{M}$)
  return oclTranslation(q_1, `OCL-Bool`$_2$, $\mathcal{M}$)
else if $q_c = q_1 \setminus q_2$ **then**
  $\mathcal{M}_1$:= getCompleteMap($\mathcal{M}$, q_1)
  $\mathcal{M}_2$:= getCompleteMap($\mathcal{M}$, q_2)
  ocl-eq := getInequalities($\mathcal{M}_1$, $\mathcal{M}_2$, q_1, q_2)
  `OCL-Bool'` := oclTranslation(q_2, ocl-eq, $\mathcal{M}_2$) **`implies`** `OCL-Bool`
  return oclTranslation(q_1, `OCL-Bool'`, $\mathcal{M}_1$)
else if q_c = R **then**
  $\mathcal{M}_2$:= getCompleteMap($\mathcal{M}$, R)
  return **`R.allInstances()->forAll(`**$\mathcal{M}$.getVar(`R.id`)`|` `OCL-Bool`**`)`**
end if

obtaining:

```
R.allInstances()->forAll(r |
      S.allInstances()->forAll(s | r <> s)
            implies false)
```

This boolean expression iterates through all elements `r` of `R`, it checks whether `r` is different from every `s` in `S`, and, if so, it returns false. Clearly, such OCL-boolean statement only evaluates to true iff the relational query `R \ S` evaluates to the empty set.

Algorithm 4 makes use of some auxiliary functions. Function *getCompleteMap(*$\mathcal{M}$*,* q*)* returns a copy of the map $\mathcal{M}$ but adding some new correspondences between relational attributes in q and new fresh OCL$_{\mathsf{FO}}$ variables. For instance, this function allowed us to create the new OCL free variables **`r`** and **`s`**, and map them to the relational tables `R` and `S` respectively. Function *getInequalities*($\mathcal{M}_1$, $\mathcal{M}_2$, q_1, q_2) returns a conjunction of OCL$_{\mathsf{FO}}$ variable inequalities. In particular, one inequality for each pair of OCL$_{\mathsf{FO}}$ variables that are mapped to the same i-th relational attribute in q_1, and q_2, respectively. This function allowed us to build the inequality `r <> s` in our previous example.

Theorem 6 *Let* `OCL-Bool` *be an OCL$_{\mathsf{FO}}$ boolean statement defined over a UML class diagram S_{UML}, q_c a relational algebra expression defined over the relational view of S_{UML}, and $\mathcal{M}$ a mapping from q_c attributes to OCL$_{\mathsf{FO}}$ variables in ϕ_1. Let ϕ be the OCL$_{\mathsf{FO}}$ statement such that ϕ = oclTranslation(q_c,* `OCL-Bool`*, $\mathcal{M}$) (Algorithm 4). Then, for any interpretation $\mathcal{I}$, and any substitution S, we have:*

$$\mathcal{I} \models \phi_{[S]} \ \textbf{iff} \ \forall_{S_t} \texttt{OCL-Bool}^{\mathcal{I}}_{[S_t][S]} = \text{TRUE}$$

where the substitutions S_t are obtained from the context query q_c and the mapping $\mathcal{M}$.

Proof The proof is inductive on the number of relational algebra operators present in the input context query q, thus, following the recursive nature of the Algorithm.

For the base case, consider a context query with the following form, where R is the name of some relation:

$$q_c = \mathtt{R}$$

Applying Algorithm 4 we get:

$$\phi = \mathtt{R}.\mathtt{allInstances()\text{->}forAll(r} \mid \mathtt{OCL\text{-}Bool)}$$

Then, according to the semantics, for any substitution S, we have that $\mathcal{I} \models \mathtt{R.allInstances()\text{->}forAll(r} \mid \mathtt{OCL\text{-}Bool)}_{[S]}$ if and only if it does not exists a value $v \in \mathtt{R}^I$ s.t. $\mathtt{OCL\text{-}Bool}^I_{[\mathtt{r}/v][S]} = \text{FALSE}$. This is the case iff for all the values $v \in \mathtt{R}^I$ we have $\mathtt{OCL\text{-}Bool}_r{}^I_{[\mathtt{r}/v][S]} = \text{TRUE}$. Equivalently, this is the case iff for all the possible substitutions S_t we can obtain from R, we have that $\mathtt{OCL\text{-}Bool}^I_{[S_t][S]} = \text{TRUE}$.

We deal now with the inductive cases. Consider first:

$$q_c = q_1 \cup q_2$$

Applying Algorithm 4 we get:

$$\phi = \mathtt{OCL\text{-}Bool}_1 \ \mathbf{and} \ \mathtt{OCL\text{-}Bool}_2$$

where

$$\mathtt{OCL\text{-}Bool}_i = \text{oclTranslation}(q_i, \mathtt{OCL\text{-}Bool}, \mathcal{M})$$

According to the semantics, $\mathcal{I} \models \phi_{[S]}$ if and only if $\mathtt{OCL\text{-}Bool}_1{}^I_{[S]} = \text{TRUE}$ and $\mathtt{OCL\text{-}Bool}_2{}^I_{[S]} = \text{TRUE}$. By induction we know that $\mathtt{OCL\text{-}Bool}_i{}^I_{[S]} = \text{TRUE}$ iff for every substitution S_t that can be obtained from q_i we have $\mathtt{OCL\text{-}Bool}^I_{[S_t][S]} = \text{TRUE}$. Thus, $\mathcal{I} \models \phi_{[S]}$ if and only if $\mathtt{OCL\text{-}Bool}^I_{[S_t][S]} = \text{TRUE}$ for every substitution that can be obtained from $q_1 \cup q_2$.

Consider now the case:

$$q_c = \pi q_1$$

Applying Algorithm 4 we get:

$$\phi = \mathtt{OCL\text{-}Bool}_1 = \text{oclTranslation}(q_1, \mathtt{OCL\text{-}Bool}, \mathcal{M})$$

And we have that $\mathcal{I} \models \phi_{[S]}$ iff $\mathtt{OCL\text{-}Bool}_1{}^I_{[S]} = \text{TRUE}$. By induction we know that $\mathtt{OCL\text{-}Bool}_1{}^I_{[S]} = \text{TRUE}$ iff for every substitution S_t that can be obtained from q_1 we have $\mathtt{OCL\text{-}Bool}^I_{[S_t][S]} = \text{TRUE}$. Thus, $\mathcal{I} \models \phi_{[S]}$ if and only if $\mathtt{OCL\text{-}Bool}^I_{[S_t][S]} = \text{TRUE}$ for every substitution that can be obtained from πq_1.

Consider the case:

$$q_c = \sigma_{a\omega b} q_1$$

Applying Algorithm 4 we get:

$$\phi = \text{oclTranslation}(q_1, \mathtt{OCL\text{-}Bool'}, \mathcal{M}')$$

where

$$\begin{aligned} \mathtt{OCL\text{-}Bool'} &= \text{`}\mathbf{v}_a \ \omega \ \mathbf{v}_b \ \mathbf{implies}\text{'} + \mathtt{OCL\text{-}Bool} \\ \mathtt{v}_a &= \mathcal{M}'.getVar(\mathtt{q_1.a}) \\ \mathtt{v}_b &= \mathcal{M}'.getVar(\mathtt{q_1.b}) \\ \mathcal{M}' &= getCompleteMap(\mathcal{M}, \mathtt{q_1}) \end{aligned}$$

By induction we know that $\mathcal{I} \models \phi_{[S]}$ iff for every substitution S_t obtained from q_1 it hdolds that $\mathtt{OCL\text{-}Bool'}^I_{[S_t][S]} = \text{TRUE}$. Unfolding OCL-Bool' we get: $\mathcal{I} \models \phi_{[S]}$ iff for every substitution S_t obtained from q_1 it holds that $\mathbf{v}_a \ \omega \ \mathbf{v}_b$ **implies** $\mathtt{OCL\text{-}Bool}^I_{[S_t][S]} = \text{TRUE}$. According to the OCL_{FO} semantics, we know that for every substitution S_t obtained from $\sigma_{a\omega b} q_1$, it holds that $\mathbf{v}_a \ \omega \ \mathbf{v}_b{}^I_{[S_t][S]} = \text{TRUE}$. Thus, $\mathcal{I} \models \phi_{[S]}$ iff for every substitution S_t obtained from $\sigma_{a\omega b} q_1$ we have $\mathtt{OCL\text{-}Bool}^I_{[S_t][S]} = \text{TRUE}$.

Consider the case:

$$q_c = q_1 \times q_2$$

Applying Algorithm 4 we get:

$$\phi = \text{oclTranslation}(q_1, \mathtt{OCL\text{-}Bool}_2, \mathcal{M})$$

where

$$\mathtt{OCL\text{-}Bool}_2 = \text{oclTranslation}(q_2, \mathtt{OCL\text{-}Bool}, \mathcal{M})$$

By induction we know that $\mathcal{I} \models \phi_{[S]}$ iff for every substitution S_{t1} from q_1 we have $\mathtt{OCL\text{-}Bool}_2{}^I_{[S_{t1}][S]} = \text{TRUE}$. By induction again, we see that $\mathcal{I} \models \phi_{[S]}$ iff for every substitution S_{t1} from q_1, and every substituition S_{t2} from q_2, we have $\mathtt{OCL\text{-}Bool}^I_{[S_{t2}][S_{t1}][S]} = \text{TRUE}$. Taking into account that any substitution S_t from q_c is obtained from any pair of substitutions S_{t1} and S_{t2} from q_1, and q_2 (respectively), we finally get $\mathcal{I} \models \phi_{[S]}$ iff for every substitution S_t from q_c we have $\mathtt{OCL\text{-}Bool}^I_{[S_t][S]} = \text{TRUE}$.

Consider the case:

$$q_c = q_1 \setminus q_2$$

Applying Algorithm 4 we get:

$$\phi = \text{oclTranslation}(q_1, \mathtt{OCL\text{-}Bool'}, \mathcal{M}_1)$$

where

$$\begin{aligned} \mathtt{OCL\text{-}Bool'} &= \text{oclTranslation}(q_2, \mathtt{ocl\text{-}ineq} + \text{`}\mathbf{implies}\text{'} + \\ &\qquad \mathtt{OCL\text{-}Bool}, \mathcal{M}_2) \\ \text{ocl-ineq} &= \text{getInequalities}(\mathcal{M}_1, \mathcal{M}_2, q_1, q_2) \\ \mathcal{M}_1 &= \text{getCompleteMap}(\mathcal{M}, q_1) \\ \mathcal{M}_2 &= \text{getCompleteMap}(\mathcal{M}, q_2) \end{aligned}$$

By induction we know that $\mathcal{I} \models \phi_{[S]}$ iff for any substitution S_t obtained from q_1 it holds that `OCL-Bool'`$^I_{[S_t][S]}$ = TRUE. Unfolding `OCL-Bool'` using the induction hypothesis we have that $\mathcal{I} \models \phi_{[S]}$ iff for any substitution S_t from q_1, and any substitution S_{t2} from q_2, it holds that (`ocl-ineq` **`implies`** `OCL-Bool`)$^I_{[S_{t2}][S_t][S]}$ = TRUE. Equivalently, $\mathcal{I} \models \phi_{[S]}$ iff for any substitution S_t from q_1, and any substitution S_{t2} from q_2, it holds that `ocl-ineq`$^I_{[S_{t2}][S_t][S]}$ = FALSE or `OCL-Bool`)$^I_{[S_t][S]}$ = TRUE. Since we know that, for any substitution S_t obtained from $q_1 \backslash q_2$ there is no substitution S_{t2} obtained from q_2 for which `ocl-ineq`$^I_{[S_{t2}][S_t][S]}$ = FALSE, we see that , $\mathcal{I} \models \phi_{[S]}$ iff for any substitution S_t obtained from $q_1 \setminus q_2$ we have `OCL-Bool`)$^I_{[S_t][S]}$ = TRUE. □

With this algorithm at hand, we now proof that any constraint that can be checked by means of relational algebra can be written in OCL$_{\mathsf{FO}}$.

Theorem 7 *Let S_{UML} be a UML class diagram, and q a relational algebra expression defined over the relational view of S_{UML}. Then, consider the OCL$_{\mathsf{FO}}$ constraint ϕ s.t. ϕ = oclTranslation(q, **`false`**, $\emptyset$) (Algorithm 4). Then, for any interpretation $\mathcal{I}$, we have:*

$$\mathcal{I} \models \phi^I \ \textbf{iff} \ q(\mathcal{I}) = \emptyset$$

Proof From Theorem 6 we know that $\mathcal{I} \models \phi^I$ **iff** for every substitution S_t obtained from q, **`false`**$^I_{S_t}$ = TRUE. This is the case if and only if there is no substitution S_t obtained from q. That is $\mathcal{I} \models \phi^I$ **iff**$q(\mathcal{I}) = \emptyset$. □

It is worth noting that the translation from RA to OCL$_{\mathsf{FO}}$ is quadratic with the number of RA operations. Indeed, most RA operators are translated into a fixed number of OCL operations (1 in most cases, 2 in the worst one), but the set-difference operator gives raise to k OCL equality operations, where k is the given number of columns of the relation where the difference is applied. Thus, the length of the translation of an RA query into OCL can be bounded by $2o + sKo$, where o is the number of RA operators, s the number of set-difference operators, and K the maximum number of columns for the biggest table. It is easy to see also that the algorithm obtains this result in quadratic time since it visits every RA operator only once, and each RA operator is translated in constant time (if no subroutine is applied), or linear time (otherwise).

7 OCL$_{\mathsf{CORE}}$

To conclude our analysis, we identify a minimal subset of OCL$_{\mathsf{FO}}$ that we call OCL$_{\mathsf{CORE}}$. OCL$_{\mathsf{CORE}}$ is minimal in the sense that any of its proper subsets is not sufficient to encode the whole OCL$_{\mathsf{FO}}$. Several minimal core fragments might exist.

The main practical relevance of OCL$_{\mathsf{CORE}}$ is that of allowing to compare the expressive power of other fragments of OCL that can be proposed in the future with that of OCL$_{\mathsf{FO}}$. Note that it is guaranteed that if a fragment contains the six operations in OCL$_{\mathsf{CORE}}$ then its expressive power will be at least that of OCL$_{\mathsf{FO}}$. For instance, using the OCL$_{\mathsf{CORE}}$ it is easy to realize that the work presented in [24] implements the whole expressiveness of OCL$_{\mathsf{FO}}$ since it implements the six basic operations. This is much easier to be determined than having to check whether the fragment contains all the operations in OCL$_{\mathsf{FO}}$.

Moreover, OCL$_{\mathsf{CORE}}$ could be used to define a UML base model [25]. That is, a minimal subset of UML/OCL features that do not loose expressiveness. In any case, we would like to stress that OCL$_{\mathsf{CORE}}$ is not intended to be used by the designers when specifying their models, since this would result in very complex and difficult to understand expressions for defining the integrity constraints, but for OCL researchers and developers.

In Figure 7 we define OCL$_{\mathsf{CORE}}$. which only contains the operations: **`allInstances`**, **`forAll`**, **`implies`**, **`<`**, and **`=`**, together with the operation to navigate from a variable to a role/attribute.

```
OCL-Bool      ::=  OCL-Bool implies OCL-Bool |
                   OCL-Set->forAll(Var | OCL-Bool) |
                   OCL-Object = OCL-Object |
                   OCL-Value < OCL-Value
OCL-Set       ::=  Class.allInstances()
OCL-Object    ::=  Var |
                   Var.role
OCL-Value     ::=  Var.fAttr |
                   ⟨a constant name⟩
```

Fig. 7 Syntax of OCL$_{\mathsf{CORE}}$

Next theorem proves that this fragment is able to encode all constraints in OCL$_{\mathsf{FO}}$.

Theorem 8 *For any OCL$_{\mathsf{FO}}$ constraint ϕ, there is an OCL$_{\mathsf{CORE}}$ constraint ϕ_c such that, for any interpretation $\mathcal{I}$:*

$$\mathcal{I} \models \phi \ \textbf{iff} \ \mathcal{I} \models \phi_c$$

Proof Given any ϕ constraint written in OCL$_{\mathsf{FO}}$, we can obtain its corresponding constraint ϕ_c written in OCL$_{\mathsf{CORE}}$ by first translating ϕ to a RA query q, and then, translating q to OCL$_{\mathsf{CORE}}$. Any OCL$_{\mathsf{FO}}$ constraint can be translated into a relational algebra query q following the process described in Section 5. Then, we

can translate q back into OCL using the process described in Section 6, thus, obtaining an $\mathrm{OCL}_{\mathsf{FO}}$ constraint ϕ'. By construction, this ϕ' already accommodates the $\mathrm{OCL}_{\mathsf{CORE}}$ syntax described in 7, except for operations **and** and **not** that might appear in ϕ' but are not included in $\mathrm{OCL}_{\mathsf{CORE}}$. However, we can easily get rid of these operations using common boolean equivalences with **implies** (e.g. **not** `OCL-Bool` is equivalent to `OCL-Bool` **implies 1=2**). □

Now, it only lacks to show that $\mathrm{OCL}_{\mathsf{CORE}}$ is minimal since we cannot remove any operation from it without loosing expressiveness. We cannot remove **allInstances** since it is the only operation that allows obtaining a set of instances from a UML class. We cannot take out **forAll** because it is the only operation that can be applied after **allInstances**. The **implies** operation is mandatory to encode, for instance, the OCL **not**. Finally, without **<**, or **=**, we would not be able to encode **<=** (among others).

It is worth noting that $\mathrm{OCL}_{\mathsf{CORE}}$ is a proper subset of normalized $\mathrm{OCL}_{\mathsf{FO}}$ which, in turn, is a proper subset of $\mathrm{OCL}_{\mathsf{FO}}$. However, from these three fragments, only $\mathrm{OCL}_{\mathsf{FO}}$ is intended for the use of the designers when specifying UML class diagrams with OCL constraints. It is worth mentioning also that the three variants have the same expressive power since both the normalized expression and the $\mathrm{OCL}_{\mathsf{CORE}}$ expression of a constraint can be automatically drawn from its $\mathrm{OCL}_{\mathsf{FO}}$ expression.

8 Related Work

We first discuss related work that studied the relationship between OCL and RA. Then we compare our OCL fragment with those fragments identified for the problem of OCL satisfiability. Finally, we compare our work with other suggested OCL translations.

8.1 OCL and Relational Algebra Studies

The expressiveness of OCL and its relationship with Relational Algebra has been previously discussed by Mandel and Cengarle in [6]. However, whereas Mandel and Cengarle addressed the expressive power of OCL as a query language, we look at OCL as a constraint language. The main difference is that a constraint language deals only with boolean expressions, so when looking at the equivalence of OCL w.r.t. RA, we focus on whether we can check some OCL constraint through checking the emptiness of a RA query q, and viceversa. In contrast, Mandel and Cengarle investigate whether for every RA query q we can build an OCL expression that returns the same tuples as q. In particular, the authors argue that this is impossible since (1) OCL has no tuple constructor, and (2) OCL has no way to dynamically create new types. However, in our equivalence notion of OCL with RA, tuples do not play a role. Indeed, we only need to check whether a true/false OCL expression can be mapped to an empty/non-empty RA query, whereas they tried to map any RA query into an equivalent OCL expression. In addition, Mandel and Cengarle hinted (although not formally proved) that OCL version 1 was not Turing-Complete. This claim has been proved to be no longer valid for OCL version 2.4 in our article.

Since OCL 2.0 introduced tuple facilities, Balsters argued that OCL is able to encode any RA query composed of the basic RA operations [26], that is: union, difference, product, renaming, selection and projection operations. However, Balsters stressed in his work that, still, OCL is not equivalent to RA in a *maximal sense* since it is impossible in OCL to define a new operation that receives as input two arbitrary sets of tuples, and outputs the natural join of them. Note that this supposed impediment does not affect us since our goal is to show that any given RA query can be rewritten into an equivalent $\mathrm{OCL}_{\mathsf{FO}}$ (not necessarily in a generic way).

From a more practical point of view, Queralt and Teniente proposed in [27] a translation from OCL to domain-independent first-order logics, which is equivalent to relational algebra. The fragment covered in their translation is expressively equivalent to $\mathrm{OCL}_{\mathsf{FO}}$ since they cover $\mathrm{OCL}_{\mathsf{CORE}}$. Interestingly, their translation is also based on first normalizing an OCL constraint into another one composed of less expressions. Probably, a further study of such normalization might bring another $\mathrm{OCL}_{\mathsf{CORE}}$ for $\mathrm{OCL}_{\mathsf{FO}}$. However, since their intention was to apply first-order reasoners on OCL rather than dicussing OCL expressiveness, they did not prove that domain-independent first-order logic statements were expressible in OCL, neither that such normalization could bring a core, as we have done.

Another translation of a fragment of OCL into first-order logic is proposed by Clavel et al. in [28]. By naturally extending their translation of inequalities to inequalities with objects, we can see that their OCL fragment covered is expressively equivalent to $\mathrm{OCL}_{\mathsf{FO}}$. In constrast, the translation given by Beckert et al. in [29] seems to deal with a broader subset of OCL. However, their translation is not pure first-order logics since, for instance, it uses some built-in functions to count the number of times an object appears in a collection unrestrictedly, which is not a first-order capability.

It is important to note that none of these proposals departs from an OCL formal semantics. Thus, none

of the previous translations has a proof of soundness. It can be argued that no soundness proof is required since, in the absence of formal OCL semantics, the semantics of OCL turns to be the translation itself. However, using such translations as the semantics for OCL is cumbersome and error prone. Indeed, they are defined by means of multiple algorithms and functions. Thus, it is extraordinarily difficult to assess, for instance, whether the semantics given by Queralt and Teniente [27] is equivalent to the one given by Clavel et al. [28]. In contrast, the OCL_{FO} semantics we provide in this paper is concise and based on basic set theory (i.e., set inclusion, exclusion, etc). Then, it could be used to prove that some translation is *sound* w.r.t. OCL_{FO} semantics, and thus, two OCL translations would be equivalent if they are both sound with respect to OCL_{FO} semantics.

Finally, there are some tools that implement translations from OCL into SQL. Egea et al. introduced MySQL4OCL[30], which generates MySQL code for a subset of OCL expressions. However, the translation defined clearly falls out of RA since it uses MySQL specific procedures. Another tool, part of the well-known Dresden OCL Toolkit [10], is OCL2SQL. It produces a translation in standard SQL, but lacks theoretical basis for sophisticated cases. Indeed, the translation is based on some straightforward patterns without any formal proof [31], thus, it is not clear the correctness of the translation when dealing with, for instance, NULL values. Indeed, OCL2SQL makes use of SQL NOT EXISTS expressions which is known to have spurious behavior when dealing with NULL values, but no discussion on this aspect is given.

8.2 OCL subsets for satisfiability

OCL_{FO} is a fragment of OCL defined for ensuring efficient integrity checking. However, other fragments of OCL have been defined pursuing different objectives. One of the most prominent is finding OCL subsets for which the problem of OCL constraint satisfiability is decidable.

The problem of constraint satisfiability is the problem of ensuring whether there exists at least one instance of the UML class diagram such that satisfies the constraint. Since it is known that checking OCL constraints satisfiability is undecidable in general [8], several decidable OCL subsets have been identified. However, since the problem of constraint satisfiability is harder than the problem of constraint checking, those OCL subsets tend to be less expressive.

In our particular case, we have that OCL_{FO} is expressively equivalent to first-order logics. Thus, since it is known that checking the satisfiability of a first-order constraint is not decidable, OCL based languages for reasoning satisfiability cannot deal with full first-order logic expressiveness. Two examples of such languages are: OCL-Lite [17], and OCL_{UNIV} [32].

OCL-Lite is a proper subset of OCL_{FO} expressively equivalent to the $\mathcal{ALCI}$ Description Logics. Essentially, it is the part of OCL_{FO} that excludes any kind of OCL operator that could be used to emulate maximum cardinalities (e.g. `->size() < k`). Indeed, to ensure that OCL-Lite constraint satisfiability is decidable, OCL-Lite must be applied to UML class diagrams without maximum cardinalities.

OCL_{UNIV} is also a proper subset of OCL_{FO}. In this case, it is based on the language of weakly acyclic TGDs [33]. The rational behind OCL_{UNIV} is that it excludes any kind of OCL operator that could be used to emulate minimum cardinalities (e.g. `->notEmpty()`, `->size() > k`). This is because OCL_{UNIV} decidability is only ensured over UML class diagrams with no minimum cardinality cycles.

Thus, it seems that, to ensure decidability for the satisfiability problem, one has to give up either *maximum cardinalities*, or *minimum cardinalities*. However, both can be easily taken into account for the problem of checking constraints (indeed, OCL_{FO} can encode both).

Figure 8 summarizes the relationship between OCL_{FO}, OCL-Lite, and OCL_{UNIV}.

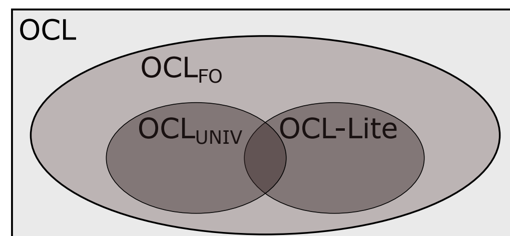

Fig. 8 Relation of different OCL fragments

8.3 Other OCL translations

There are some tools in the literature that translates OCL into other languages. These tools are mainly aimed to the problem of OCL constraint satisfiability, and ensure its decidability by means of a bounding the search space (e.g. limiting the size of the UML instance to search for).

For instance, the approaches presented in [34–36] proposes a translation from OCL into the Alloy/Kodkod languages so that, satisfiability can be checked through the Alloy/Kodkod reasoner. It is worth to mention that these tools can deal with the OCL `->closure` operation, which is outside first-order logics, and thus, outside OCL_{FO}. Furthermore, more ex-

pressive translations based on Constraint-Satisfaction Problem [37], and Satisfiability Modulo Theories [38] have also been proposed. An easy way to see that these translations are more expressive is that, on one hand, they implment all the operations from $\mathrm{OCL}_{\mathsf{CORE}}$, and they provide some operations not inside $\mathrm{OCL}_{\mathsf{FO}}$ (such as the comparison of the size of two sets, which is not a first-order expression).

However, these approaches are only focused on reasoning constraint satisfiability, and are not though for bringing formal semantics into OCL, neither showing efficient satisfaction checking, as we do.

Indeed, we bring formal semantics based on set-theory. These semantics describe the evaluation of OCL expressions in a brief and unambiguous manner as can be seen in Figure 5. Thus, these semantics can be used to check/proof the correctness of OCL translations (as we do with the translation between OCL and RA). However, the previous translations are not checked/proved on any particular OCL semantics. Thus, it is impossible to know how does such translations interpret OCL expressions (neither if they are pairwise equivalent) without studying the semantics of the particular language they use, which are not even standard in the case of Alloy/Kodkod.

9 Conclusions

OCL is a formal language for defining constraints that serves as a complement for graphical modelling languages such as UML. However, full OCL is so expressive that checking OCL constraints is not even semidecidible, as we have shown in this paper.

To tackle this issue, we have identified $\mathrm{OCL}_{\mathsf{FO}}$, the fragment of OCL equivalent to RA. That is, any $\mathrm{OCL}_{\mathsf{FO}}$ constraint can be evaluated by checking if some RA query is empty (which guarantees efficiency), and any RA query can be translated into $\mathrm{OCL}_{\mathsf{FO}}$ (which guarantees expressiveness).

The syntax and semantics of $\mathrm{OCL}_{\mathsf{FO}}$ are defined in a concise way and thus, we argue that they can be easily adopted by OCL practitioners. Moreover, we identify the minimal subset of $\mathrm{OCL}_{\mathsf{FO}}$ with its same expressive power, $\mathrm{OCL}_{\mathsf{CORE}}$, which makes $\mathrm{OCL}_{\mathsf{FO}}$ an easy object of study.

As future work, we plan to exploit the equivalence between $\mathrm{OCL}_{\mathsf{FO}}$ and RA to adapt the techniques from the database field to the OCL and conceptual modeling community. In the first place, we would like to study an automatic efficient translation from $\mathrm{OCL}_{\mathsf{FO}}$ to SQL including optimizations such as indexes, or better encodings of the associations into relational tables. This translation could also benefit from incremental query techniques (thus, obtaining incremental integrity checking), and view updating proposals (thus, obtaining $\mathrm{OCL}_{\mathsf{FO}}$ incremental maintenance). Indeed, some proposals treating OCL problems through database techniques have recently been published with exciting results [23, 14, 16]. Furthermore, we would like to study if we can include OCL Bags, OrderedSets, and Sequences into $\mathrm{OCL}_{\mathsf{FO}}$, for instance, by means of adapting the techniques described in [39].

Acknowledgements This work has been partially supported by the Ministerio de Economía y Competitividad, under project TIN2017-87610-R, and by the Secreteria d'Universitats i Recerca de la Generalitat de Catalunya under 2017 SGR 1749.

References

1. Chen, P.P.S.: The entity-relationship model-toward a unified view of data. ACM Transactions on Database Systems (TODS) **1**(1) (1976) 9–36
2. Halpin, T.: Object-role modeling (orm/niam). In: Handbook on architectures of information systems. Springer (1998) 81–103
3. Object Management Group (OMG): Unified Modeling Language (UML) Superstructure Specification, version 2.4.1. (2011) http://www.omg.org/spec/UML/.
4. Object Management Group (OMG): Object Constraint Language (UML), version 2.4. (2014) http://www.omg.org/spec/OCL/.
5. Immerman, N.: Descriptive complexity. Springer Science & Business Media (2012)
6. Mandel, L., Cengarle, M.V.: On the expressive power of the object constraint language OCL. In: FM'99 — Formal Methods. Volume 1708 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1999) 854–874
7. Brucker, A.D., Clark, T., Dania, C., Georg, G., Gogolla, M., Jouault, F., Teniente, E., Wolff, B.: Panel discussion: Proposals for improving ocl. In: Proceedings of the MODELS 2014 OCL Workshop (OCL 2014). Volume 1285., CEUR-WS. org (2014) 83–99
8. Berardi, D., Calvanese, D., De Giacomo, G.: Reasoning on UML class diagrams. Artif. Intell. **168**(1-2) (2005) 70–118
9. : Eclipse ocl project. http://wiki.eclipse.org/OCL Accessed: 2016-08-08.
10. Aßmann, U., Bartho, A., Bürger, C., Cech, S., Demuth, B., Heidenreich, F., Johannes, J., Karol, S., Polowinski, J., Reimann, J., Schroeter, J., Seifert, M., Thiele, M., Wende, C., Wilke, C.: Dropsbox: the dresden open software toolbox. Software & Systems Modeling **13**(1) (2014) 133–169
11. Hamann, L., Hofrichter, O., Gogolla, M.: On integrating structure and behavior modeling with OCL. In: Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, 2012. Proceedings. (2012) 235–251
12. Brucker, A.D., Tuong, F., Wolff, B.: Featherweight ocl: A proposal for a machine-checked formal semantics for ocl 2.5. Archive of Formal Proofs (January 2014) http://www.isa-afp.org/entries/Featherweight_OCL.shtml, Formal proof development.
13. Marković, S., Baar, T. In: An OCL Semantics Specified with QVT. Springer Berlin Heidelberg, Berlin, Heidelberg (2006) 661–675

14. Oriol, X., Teniente, E.: Incremental checking of ocl constraints with aggregates through sql. In: Conceptual Modeling: 34th International Conference, ER, Cham, Springer International Publishing (2015) 199–213
15. Franconi, E., Mosca, A., Oriol, X., Rull, G., Teniente, E.: Logic foundations of the ocl modelling language. In: European Workshop on Logics in Artificial Intelligence, Springer (2014) 657–664
16. Oriol, X., Teniente, E., Tort, A.: Computing repairs for constraint violations in uml/ocl conceptual schemas. Data & Knowledge Engineering **99** (2015) 39–58
17. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: finite reasoning on UML/OCL conceptual schemas. Data & Knowledge Engineering **73** (2012) 1–22
18. Linz, P.: An Introduction to Formal Languages and Automata. Jones & Bartlett Learning (1990)
19. Queralt, A., Teniente, E.: Verification and validation of uml conceptual schemas with ocl constraints. ACM Trans. Softw. Eng. Methodol. **21**(2) (March 2012) 13:1–13:41
20. Planas, E., Olivé, A.: The DBLP case study (2006) http://www-pagines.fib.upc.es/~modeling/DBLP.pdf.
21. Tort, A.: The osCommerce case study http://www-pagines.fib.upc.es/~modeling/osCommerce_cs.pdf.
22. ANSI Standard: The SQL 92 Standard. (1992)
23. Bergmann, G.: Translating OCL to graph patterns. In: Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, 2014. Proceedings. (2014) 670–686
24. Egea, M., Dania, C.: Sql-pl4ocl: an automatic code generator from ocl to sql procedural language. Software & Systems Modeling (May 2017)
25. Hilken, F., Niemann, P., Gogolla, M., Wille, R.: From UML/OCL to base models: Transformation concepts for generic validation and verification. In: Theory and Practice of Model Transformations - 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings. (2015) 149–165
26. Balsters, H.: Modelling database views with derived classes in the UML/OCL-framework. In: UML2003-The Unified Modeling Language. Modeling Languages and Applications. Springer (2003) 295–309
27. Queralt, A., Teniente, E.: Verification and validation of UML conceptual schemas with OCL constraints. ACM Trans. Softw. Eng. Methodol. **21**(2) (2012) 13
28. Clavel, M., Egea, M., de Dios, M.A.G.: Checking unsatisfiability for OCL constraints. In: Proceedings of the Workshop The Pragmatics of OCL and Other Textual Specification Languages. Volume 24., ECEASST (2009)
29. Beckert, B., Keller, U., Schmitt, P.H.: Translating the object constraint language into first-order predicate logic. In: Proceedings of VERIFY, Workshop at Federated Logic Conferences (FLoC). (2002)
30. Egea, M., Dania, C., Clavel, M.: MySQL4OCL: A stored procedure-based MySQL code generator for OCL. Electronic Communications of the EASST **36** (2010)
31. Demuth, B., Hussmann, H.: Using UML/OCL constraints for relational database design. In: «UML»99 - The Unified Modeling Language. Springer (1999) 598–613
32. Oriol, X., Teniente, E.: Ocl_{univ}: Expressive UML/OCL conceptual schemas for finite reasoning. In: Conceptual Modeling - 36th International Conference, ER 2017, Valencia, Spain, November 6-9, 2017, Proceedings. (2017) 354–369
33. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: semantics and query answering. Theor. Comput. Sci. **336**(1) (2005) 89–124
34. Cunha, A., Garis, A.G., Riesco, D.: Translating between alloy specifications and UML class diagrams annotated with OCL. Software and System Modeling **14**(1) (2015) 5–25
35. Kuhlmann, M., Gogolla, M.: From UML and OCL to relational logic and back. In: Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings. (2012) 415–431
36. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to alloy. Software and System Modeling **9**(1) (2010) 69–86
37. González, C.A., Büttner, F., Clarisó, R., Cabot, J.: Emftocsp: a tool for the lightweight verification of EMF models. In: Proceedings of the First International Workshop on Formal Methods in Software Engineering - Rigorous and Agile Approaches, FormSERA 2012, Zurich, Switzerland, June 2, 2012. (2012) 44–50
38. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL data types for sat-based verification of UML/OCL models. In: Tests and Proofs - 5th International Conference, TAP 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings. (2011) 152–170
39. Kuhlmann, M., Gogolla, M.: Strengthening sat-based validation of UML/OCL models by representing collections as relations. In: Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings. (2012) 32–48