OCL-Lite: A Decidable (Yet Expressive) Fragment of OCL

Anna Queralt\textsuperscript{2}, Alessandro Artale\textsuperscript{1}, Diego Calvanese\textsuperscript{1}, and Ernest Teniente\textsuperscript{2}

\textsuperscript{1} KRDB Research Centre for Knowledge and Data
Free University of Bozen-Bolzano, Italy
\{artale,calvanese\}@inf.unibz.it

\textsuperscript{2} Dept. of Service and Information System Engineering
UPC – BarcelonaTech, Spain
\{aqueralt,teniente\}@essi.upc.edu

Abstract. UML has become a de facto standard in conceptual modeling. Class diagrams in UML allow one to model the data in the domain of interest by specifying a set of graphical constraints. However, in most cases one needs to provide the class diagram with additional semantics to completely specify the domain, and this is where OCL comes into play. While reasoning over class diagrams is decidable and has been investigated intensively, it is well known that checking the correctness of OCL constraints is undecidable. Thus, we introduce OCL-Lite, a fragment of the full OCL language and prove that reasoning over UML class diagrams with OCL-Lite constraints is in \textit{ExpTime} by an encoding in the description logic \textit{ALCI}. As a side result, DL techniques and tools can be used to reason on UML class diagrams annotated with arbitrary OCL-Lite constraints.

1 Introduction

The Unified Modeling Language (UML) has become a de facto standard in conceptual modeling of information systems. In UML, a conceptual schema is represented by means of a class diagram, with its graphical constraints, together with a set of user-defined constraints, which are usually specified in the Object Constraint Language (OCL). In every application domain the set of instances that can be stored or managed is necessarily finite and, thus, a schema has not only to be consistent, but also finitely satisfiable [13]. It is well-known that reasoning with OCL integrity constraints in their full generality is undecidable since it amounts to full FOL reasoning [4]. Thus, reasoning with UML conceptual schemas in the presence of OCL constraints has been approached in the following alternative ways:

\* This paper is a shortened version of [14]. This work has been partly supported by the Ministerio de Ciencia e Innovación under the projects TIN2008-03863 and TIN2008-00444, Grupo Consolidado.
1. Allowing general constraints (in OCL or other languages) without guaranteeing termination in all cases [9, 6, 15].
2. Allowing general constraints and ensuring termination without guaranteeing completeness of the result [1, 17, 12].
3. Ensuring both termination and completeness of finite reasoning by allowing only specific kinds of constraints [13, 11, 18].
4. Ensuring terminating and complete reasoning by disallowing OCL constraints and admitting unrestricted models [8, 5, 10, 3, 2].

To our knowledge, none of the existing approaches guarantees complete and terminating reasoning on UML schemas coupled with OCL constraints able to capture those in Figure 1. With approaches of the first kind, a result may not be obtained in some particular cases. On the contrary, the second kind of approaches may fail to find existing valid solutions. Approaches of the third kind do not allow arbitrary constraints as the ones in Figure 1. Finally, the last approaches are based on a Description Logic (DL) encoding of a UML schema. These methods do not consider OCL constraints and they usually check unrestricted satisfiability.

The main purpose of this paper is to identify a fragment of OCL, which we call OCL-Lite, that guarantees finite satisfiability while being significantly expressive at the same time. In other words, we propose the specification of arbitrary constraints within the bounds of OCL-Lite in a UML conceptual schema to ensure completeness and decidability of reasoning. Such nice properties are due to the finite model property (FMP) of the language, i.e., it is guaranteed that a satisfiable UML/OCL-Lite schema is always finitely satisfiable. The proposed OCL-Lite is the result of identifying a fragment of OCL that can be encoded into the DL ALCI [7], which has interesting reasoning properties. In particu-
lar, \textit{ALCI} enjoys the FMP, i.e., every satisfiable set of constraints formalized in \textit{ALCI} admits a finite model. Thus, the FMP is also guaranteed for any fragment of OCL that fits into \textit{ALCI}.

The contributions of this paper can be summarized as follows:

- The identification of a fragment of OCL that enjoys the FMP. To our knowledge, the reasoning properties of OCL had not been studied before, except that full OCL leads to undecidability.
- A mapping from OCL-Lite to the DL \textit{ALCI} to prove that the proposed fragment has the same reasoning properties as \textit{ALCI}. To our knowledge, this is the first attempt to encode OCL constraints in DLs. As a side result, a DL reasoner can be used to verify the correctness of a schema.
- Thanks to the mapping to \textit{ALCI} we are able to show both the FMP and that checking satisfiability in UML/OCL-Lite is an ExPTIME-complete problem.

2 OCL-Lite Syntax and Semantics

In this section we present the fragment of OCL that corresponds to OCL-Lite. We start by an overview of basic concepts of UML and OCL.

A UML class diagram represents the static view of the domain basically by means of classes and associations between them, representing, respectively, sets of objects and relations between objects. An association is constrained by a set of participating classes. An association end is a part of an association that defines the participation of a class in the association. The name of the role played by a participant in an association is placed in the association end near the corresponding class. If the role name is omitted, the role played by the participant is the name of its class.

An OCL constraint (or invariant) has the form: context $C$ inv: $OCLExpr$, where $C$ is the contextual class, i.e., the class to which the constraint belongs, and $OCLExpr$ is an expression that results in a boolean value. An OCL constraint is satisfied by an instantiation of the schema if $OCLExpr$ evaluates to true for each instance of the contextual class. An OCL expression is defined by means of navigation paths, combined with predefined OCL operations to specify conditions on those paths. A navigation path is a sequence of role names defined in the associations of the class diagram. Each role name used in a path indicates the direction of the navigation.

2.1 OCL-Lite Syntax

In this section we specify the syntax rules that allow one to construct OCL constraints belonging to the fragment of OCL that we call OCL-Lite. An OCL-Lite constraint has the form: context $C$ inv: $OCL-LiteExpr$. In the syntax rules, shown in Figure 2, an OCL-Lite expression $OCL-LiteExpr$ is defined recursively. Intuitively, $OCL-LiteExpr$ allows one to construct a boolean OCL-Lite expression, which can correspond to the whole constraint, or can be the condition
**OCL-LiteExpr** ::= Path SelectExpr | oclIsTypeOf(C) | not OCL-LiteExpr |
OCL-LiteExpr and OCL-LiteExpr |
OCL-LiteExpr or OCL-LiteExpr |
OCL-LiteExpr implies OCL-LiteExpr

**Path** ::= PathItem | PathItem.Path

**PathItem** ::= r_i | oclAsType(C).r_i

**SelectExpr** ::= BooleanOp | ->select(OCL-LiteExpr) SelectExpr

**BooleanOp** ::= ->exists(OCL-LiteExpr) | ->forall(OCL-LiteExpr) | ->size()>0 | ->size()=0 | ->isEmpty() | ->notEmpty()

Fig. 2. Syntax of OCL-Lite expressions

specified as a parameter in a select operation (see SelectExpr) or in exists and forall operations (see BooleanOp). An OCL-LiteExpr can be either a Path to which a SelectExpr is applied, a check of whether an object is of a certain type, or boolean combinations of these OCL-Lite expressions.

The label Path indicates how a navigation path can be constructed as a non-empty sequence of PathItems. Each PathItem can be either a role name r_i specified in the class diagram, or a role name preceded by the operation oclAsType(C), when we need to access a role name of a particular class C. When OCL-LiteExpr corresponds to the whole constraint, each path starts from a role that is accessible from the contextual class (or a subclass C of the contextual class, in which case oclAsType(C) must be specified first). Otherwise, when OCL-LiteExpr is inside a select, exists, or forall operation, then, the starting role name will depend on the context where the operation is used. After a Path, one can apply a (possibly empty) sequence of selections on the collection of objects obtained through the path, always followed by a BooleanOp.

The intuitive meaning of the OCL collection operations included in this fragment of OCL is as follows. Let col denote the collection of objects reachable along a path, and let o denote a single object, then:

- col->select(OCL-LiteExpr): returns the subset of elements of col that satisfy OCL-LiteExpr;
- col->exists(OCL-LiteExpr): returns true iff there is some element of col that satisfies OCL-LiteExpr;
- col->forall(OCL-LiteExpr): returns true iff all the elements of col satisfy OCL-LiteExpr;
- col->size(): returns the number of elements of col;
- col->isEmpty(): returns true iff col is empty;
- col->notEmpty(): returns true iff col is not empty;
- o.oclIsTypeOf(C): returns true iff o is an instance of the class C;

All constraints in Figure 1 are examples of well-formed OCL-Lite expressions.

### 2.2 OCL-Lite Semantics

OCL-Lite operations, except for oclIsTypeOf and oclAsType, can be expressed only in terms of select, isEmpty, and notEmpty. Thus, to specify the seman-
Table 1. Normalization of OCL-Lite expressions

<table>
<thead>
<tr>
<th>Original expression</th>
<th>Normalized expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) col-&gt;exists(cond)</td>
<td>col-&gt;select(cond)-&gt;notEmpty()</td>
</tr>
<tr>
<td>b) col-&gt;forall(cond)</td>
<td>col-&gt;select(not cond)-&gt;isEmpty()</td>
</tr>
<tr>
<td>c) col-&gt;select(cond1)-&gt;select(cond2)</td>
<td>col-&gt;select(cond1 and cond2)</td>
</tr>
<tr>
<td>d) col-&gt;size()&gt;0</td>
<td>col-&gt;notEmpty()</td>
</tr>
<tr>
<td>e) col-&gt;size()=0</td>
<td>col-&gt;isEmpty()</td>
</tr>
<tr>
<td>f) not col-&gt;isEmpty()</td>
<td>col-&gt;notEmpty()</td>
</tr>
<tr>
<td>g) not col-&gt;notEmpty()</td>
<td>col-&gt;isEmpty()</td>
</tr>
</tbody>
</table>

tics of OCL-Lite expressions, we first rewrite each expression as an equivalent normalized one, which is expressed in terms of these operations only. Table 1 shows the OCL-Lite expressions together with their normal form. These normalizations, together with de Morgan’s laws, are iteratively applied until the expression only contains the operations select, isEmpty, and notEmpty, and the boolean operator not only appears before oclIsTypeOf(C). In the table, col and cond denote, respectively, a collection and a condition, which must be defined according to the syntax rules.

In our running example, the constraints in Figure 1 that have to be normalized are 1, 3, 4, and 5. The resulting expressions we get after applying the rules in Table 1 are:

– Constraint 1. We apply rule a) and we get the normalized expression:

context Person inv:

organized->select(oclIsTypeOf(CriticalEvent) and sponsor->isEmpty())->notEmpty()

– Constraint 3. We first apply rule b) and we get:

context Person inv:

audited->select(not sponsor->notEmpty())->isEmpty()

We then apply rule g) to obtain the normalized expression:

context Person inv:

audited->select(sponsor->isEmpty())->isEmpty()

– Constraint 4. We apply rule f) and we get:

context CriticalEvent inv: responsible.organized->notEmpty()

– Constraint 5. We apply rule a) and we get:

context Sponsor inv:

event->select(oclIsTypeOf(CriticalEvent))->notEmpty() or event->select(sibling->isEmpty() or sibling->select(sponsor->isEmpty())->notEmpty())->notEmpty()

It can be seen from Table 1 that the expressions resulting from the normalization conform to a limited set of patterns, basically consisting of an optional select operation followed either by isEmpty or notEmpty. Also, the expression may include the operation oclIsTypeOf.

In the following we consider OCL-Lite expressions in their normal form. For each of them we specify its semantics by means of an interpretation function, f,
that maps each $OCL\text{-LiteExpr}$ into a FOL formula $OCL\text{-LiteExpr}'(x)$ with one free variable. Other approaches specify the semantics of OCL expressions using first-order terms instead of formulas [4]. However, as also argued in [4], using formulas is preferable when dealing with sets, as in our case.

We start by formalizing the semantics of an OCL-Lite constraint

$$\text{context } C \text{ inv: } OCL\text{-LiteExpr}$$

Its interpretation is: $\forall x \in C(x) \rightarrow OCL\text{-LiteExpr}'(x)$ where $C$ is the unary predicate corresponding to class $C$.

To define the semantics of OCL-Lite expressions, we first introduce some notation to deal with navigation paths. Consider a navigation path $p_1 \ldots p_i$ in an OCL-Lite expression, where each $p_i$ is either a role name or $\text{oclAsType}(C_i).r_i$. To formalize a (binary) association $A$, we introduce a binary predicate, $A_i$, whose first argument represents an instance of $\text{dom}(A_i)$ (the domain of $A_i$) and whose second argument represents an instance of $\text{range}(A_i)$ (the range of $A_i$). To fix a semantics for role names we conform to the UML convention about role names [16], i.e., a role name attached to an association end labeled with a class $C$ is used to navigate from one object to another one belonging to the class $C$. Thus, in the following, $p_i(x, y) = A_i(x, y)$ when $p_i$ is a role name attached to the $\text{range}(A_i)$-end of $A_i$, and, vice versa, $p_i(x, y) = A_i(y, x)$ when $p_i$ is a role name attached to the $\text{dom}(A_i)$-end of $A_i$. Similarly, $p_i(x, y) = C_i(x) \land A_i(x, y)$, when $p_i = \text{oclAsType}(C_i).r_i$ and $r_i$ is a role name attached to the $\text{range}(A_i)$-end of $A_i$, while $p_i(x, y) = C_i(x) \land A_i(y, x)$, when $p_i = \text{oclAsType}(C_i).r_i$ and $r_i$ is a role name attached to the $\text{dom}(A_i)$-end of $A_i$.

1. $OCL\text{-LiteExpr} = p_1 \ldots p_i \rightarrow \text{select}(OCL\text{-LiteExpr}_0) \rightarrow \text{notEmpty}()$

   The semantics of this expression is

   $$OCL\text{-LiteExpr}'(x) = \exists x_1 \ldots \exists x_i (p'_1(x, x_n) \land \ldots \land p'_i(x_2, x_1) \land OCL\text{-LiteExpr}'_0(x_1))$$

   A particular case of this expression is when no $\text{select}$ operation is applied on the objects obtained from the navigation, which corresponds to the expression $OCL\text{-LiteExpr} = p_1 \ldots p_i \rightarrow \text{notEmpty}()$. In this case, we have

   $$OCL\text{-LiteExpr}'(x) = \exists x_1 \ldots \exists x_i (p'_1(x, x_n) \land \ldots \land p'_i(x_2, x_1))$$

2. $OCL\text{-LiteExpr} = p_1 \ldots p_i \rightarrow \text{select}(OCL\text{-LiteExpr}_0) \rightarrow \text{isEmpty}()$

   The semantics of this expression is

   $$OCL\text{-LiteExpr}'(x) = \forall x_1 \ldots \forall x_i (\neg p'_1(x, x_n) \lor \ldots \lor \neg p'_i(x_2, x_1) \lor \neg OCL\text{-LiteExpr}'_0(x_1))$$

   Again, we have a particular case of this kind of expression in the absence of $\text{select}$, namely $OCL\text{-LiteExpr} = p_1 \ldots p_i \rightarrow \text{isEmpty}()$. In this case, the semantics of the expression is

   $$OCL\text{-LiteExpr}'(x) = \forall x_1 \ldots \forall x_i (\neg p'_1(x, x_n) \lor \ldots \lor \neg p'_i(x_2, x_1))$$

3. $OCL\text{-LiteExpr} = [\text{not}] \text{oclIsTypeOf}(C)$

   where the brackets denote optionality. The semantics of the expression is

   $$OCL\text{-LiteExpr}'(x) = \neg C(x)$$
4. OCL-LiteExpr = OCL-LiteExpr₁ and OCL-LiteExpr₂
The semantics of this expression is
\[ \text{OCL-LiteExpr}^f(x) = \text{OCL-LiteExpr}^f₁(x) \land \text{OCL-LiteExpr}^f₂(x) \]

5. OCL-LiteExpr = OCL-LiteExpr₁ or OCL-LiteExpr₂
The semantics of this expression is
\[ \text{OCL-LiteExpr}^f(x) = \text{OCL-LiteExpr}^f₁(x) \lor \text{OCL-LiteExpr}^f₂(x) \]

6. OCL-LiteExpr = OCL-LiteExpr₁ implies OCL-LiteExpr₂
The semantics of this expression is
\[ \text{OCL-LiteExpr}^f(x) = \text{OCL-LiteExpr}^f₁(x) \rightarrow \text{OCL-LiteExpr}^f₂(x) \]

**Definition 1 (Satisfiability of OCL-Lite constraints).** Let \( \Gamma \) be a set of OCL-Lite constraints and \( \Gamma^f \) the resulting FOL theory. Then, \( \Gamma \) is said to be satisfiable if there exists a first order interpretation \( I = (\Delta^I, \cdot^I) \) that satisfies \( \Gamma^f \). \( I \) is called a model of \( \Gamma \).

### 3 Encoding UML/OCL-Lite in ALCIT

In this section we show that the proposed fragments of OCL and UML can both be encoded in the DL ALCIT. Thus, finite reasoning is guaranteed for every conceptual schema expressed in this language. We first present the fragment of UML we are interested in, and then we provide an encoding for the OCL-Lite fragment, too.

We assume the encoding of a fragment of UML class diagrams in ALCIT, based on the encoding in ALCQI proposed in [5]. Since ALCQI is an extension of ALCIT that does not have the FMP [7] (i.e., a schema specified in ALCQI might be satisfiable only by an infinite number of instances), we focus on a fragment of UML that can be encoded into ALCIT. In particular, we consider UML class diagrams where the domain of interest is represented through classes (representing sets of objects), possibly equipped with attributes and associations (representing relations among objects), and types (representing the domains of attributes, i.e., integer, string, etc.). The kind of UML constraints that we consider in this paper are the ones typically used in conceptual modeling, namely hierarchical relations between classes, disjointness and covering between classes, cardinality constraints for participation of entities in relationships, and multiplicity and typing constraints for attributes. To preserve the FMP we restrict both cardinality and multiplicity constraints to be of the form \(*\) or \(1..*\) (meaning that either no constraint applies or the class participates at-least once, respectively).

The encoding, starting from a UML class diagram \( \Sigma \), generates a satisfiability preserving ALCIT knowledge base \( K_\Sigma \) (see [5] for details on the encoding). Given the UML fragment chosen, a model of the knowledge base \( K_\Sigma \) can be viewed as an instantiation of the UML class diagram \( \Sigma \).

In the following, we provide a mapping to translate OCL-Lite constraints into ALCIT. An OCL-Lite constraint, which has the general form
context C inv: OCL-LiteExpr

is encoded as the following ALCI inclusion assertion:

\[ C \subseteq \text{OCL-LiteExpr}^\dagger \]

where \( \cdot^\dagger \) is a mapping function that assigns to each OCL-Lite expression its corresponding ALCI encoding. This inclusion assertion, according to the semantics of OCL constraints, states that each instance of \( C \) must satisfy \( \text{OCL-LiteExpr} \).

We now illustrate the encoding of OCL-Lite expressions in ALCI. We consider OCL-Lite expressions in their normal form, as provided in Section 2.2, and define \( \text{OCL-LiteExpr}^\dagger \) by induction on the structure of \( \text{OCL-LiteExpr} \).

1. \( \text{OCL-LiteExpr} = p_n \ldots p_1 \rightarrow \text{select}(\text{OCL-LiteExpr}_0) \rightarrow \text{notEmpty()} \)

   We define the ALCI concept \( \text{OCL-LiteExpr}^\dagger \) by induction on the length \( n \) of the navigation path. For convenience, we consider as base case \( n = 0 \), and in this case we set \( \text{OCL-LiteExpr}^\dagger = \text{OCL-LiteExpr}_0^\dagger \).

   For the inductive case, let \( \text{OCL-LiteExpr}_{n+1} = p_n \ldots p_1 \rightarrow \text{select}(\text{OCL-LiteExpr}_0) \rightarrow \text{notEmpty()} \), let \( p_{n+1} \) be an additional path item, and let \( \text{OCL-LiteExpr}_{n+1} = p_{n+1} \cdot \text{OCL-LiteExpr}_n \).

   Then \( \text{OCL-LiteExpr}^\dagger_{n+1} = p_{n+1} \cdot (\text{OCL-LiteExpr}_n^\dagger) \), where \( p_{n+1} \) (for the various cases of \( p_{n+1} \), cf. the OCL syntax in Section 2.1) is an abbreviation\(^3\) defined as follows (\( r \) denotes a role name of an association \( A \), and \( \text{dom}(A) \) and \( \text{range}(A) \) denote respectively the domain and range of \( A \)):

   \[
   r^\dagger = \begin{cases} 
   \exists A & \text{if } r \text{ is attached to } \text{range}(A) \\
   \exists A^- & \text{if } r \text{ is attached to } \text{dom}(A) 
   \end{cases}
   \]

   \[
   (\text{oclAsType}(C).r)^\dagger = \begin{cases} 
   C \cap \exists A & \text{if } r \text{ is attached to } \text{range}(A) \\
   C \cap \exists A^- & \text{if } r \text{ is attached to } \text{dom}(A) 
   \end{cases}
   \]

   Note that the ALCI concept corresponding to \( \text{OCL-LiteExpr} \) has the form

   \[
   p_n^\dagger \cdot (p_{n-1}^\dagger \cdot (\cdots (p_1^\dagger \cdot (\text{OCL-LiteExpr}_0^\dagger) \cdots )))
   \]

   Intuitively, this concept represents the fact that \( \text{OCL-LiteExpr} \) evaluates to true, for a given instance \( o \) in the domain of \( p_n \), if \( o \) is related through the path \( p_n \ldots p_1 \) to some object \( o_1 \) that satisfies the condition specified by \( \text{OCL-LiteExpr}_0 \). When there is no \text{select} \) operation, i.e., the OCL expression has the form \( p_n \ldots p_1 \rightarrow \text{notEmpty()} \), then \( \text{OCL-LiteExpr}_0^\dagger = \top \), and the constraint is encoded as \( p_n^\dagger \cdot (p_{n-1}^\dagger \cdot (\cdots (p_1^\dagger \cdot \top) \cdots )) \).

   For example, once it has been normalized in Section 2.2, an expression that follows this pattern is the one in the body of constraint 4. The ALCI assertion that encodes this constraint is:

   \[
   \text{CriticalEvent} \subseteq \exists \text{ResponsibleFor}^- \cdot (\exists \text{Organizes}.\top)
   \]

\(^3\) Note that \( p^\dagger \) is not a valid DL expression.
Note that, the first DL role, ResponsibleFor\(^\rightarrow\), is inverted since the association ResponsibleFor has domain Person and range CriticalEvent, and the role name responsible is attached to Person. Thus, responsible\(^\dagger\) = \exists\text{ResponsibleFor}\(^\rightarrow\). The next role name in the OCL-Lite expression is organized, which is attached to Event, the range of the association Organizes. Thus, organized\(^\dagger\) = \exists\text{Organizes}. Finally, since the OCL operation select does not appear in the constraint, no condition must be imposed on the instances reached at the end of the path.

2. OCL-LiteExpr = p\(_n\)...p\(_1\)->select(OCL-LiteExpr\(_0\))\(\rightarrow\)isEmpty()

Similarly to the previous case, we define OCL-LiteExpr\(^\dagger\) by induction on \(n\).

For the base case of \(n = 0\), we set OCL-LiteExpr\(^\dagger\) = \neg\text{OCL-LiteExpr\(_0\)}\(^\dagger\). For the inductive case, let:

\[ \text{OCL-LiteExpr\(_n\)} = p\(_n\)...p\(_1\)->select(OCL-LiteExpr\(_0\))\(\rightarrow\)isEmpty(), \]

let \(p\(_{n+1}\)\) be an additional path item, and let

\[ \text{OCL-LiteExpr\(_{n+1}\)} = p\(_{n+1}\).\text{OCL-LiteExpr\(_n\)}\]

Then OCL-LiteExpr\(_{n+1}\) = \neg\text{p\(_n\)}\(\neg\text{OCL-LiteExpr\(_n\)}\(^\dagger\)). Considering that \neg\(C\) is equivalent to \(C\), the ALCI concept corresponding to OCL-LiteExpr has the form

\[ \neg(p\(_n\)(p\(_{n-1}\)(\cdots(p\(_1\).(OCL-LiteExpr\(_0\))\cdots)))\]

Intuitively, this concept represents the fact that OCL-LiteExpr evaluates to true, for a given instance \(o\), if \(o\) is not related through the path \(p\(_n\)...p\(_1\)\) to any object that satisfies the condition specified by OCL-LiteExpr\(_0\). As in the previous case, if there is no select operation in the OCL expression, i.e., the OCL expression has the form \(p\(_n\)...p\(_1\)->isEmpty()\), then OCL-LiteExpr\(_0\)\(^\dagger\) = \top, and the constraint is encoded as \neg(p\(_n\)(p\(_{n-1}\)(\cdots(p\(_1\).\top)\cdots)))

As an example, let us consider constraint 3 in its normal form. The overall OCL-Lite expression is encoded in ALCI as \neg(\exists\text{Audits}.(\neg\exists\text{SponsoredBy}.\top)), and the ALCI assertion corresponding to the constraint is:

\text{Person} \sqsubseteq \neg\exists\text{Audits}.\neg\exists\text{SponsoredBy}.\top

3. OCL-LiteExpr = oclIsTypeOf(C), OCL-LiteExpr = not oclIsTypeOf(C)

The ALCI concept OCL-LiteExpr\(^\dagger\) corresponding to these OCL-Lite expressions is respectively

\(\top\)\(\neg\text{C}\),

4. OCL-LiteExpr = OCL-LiteExpr\(_1\) and OCL-LiteExpr\(_2\)

The corresponding ALCI concept OCL-LiteExpr\(^\dagger\) is

\[ \text{OCL-LiteExpr\(_1\)} \sqcap \text{OCL-LiteExpr\(_2\)}\]

5. OCL-LiteExpr = OCL-LiteExpr\(_1\) or OCL-LiteExpr\(_2\)

The corresponding ALCI concept OCL-LiteExpr\(^\dagger\) is

\[ \text{OCL-LiteExpr\(_1\)} \sqcup \text{OCL-LiteExpr\(_2\)}\]
6. $\text{OCL-LiteExpr} = \text{OCL-LiteExpr}_1 \implies \text{OCL-LiteExpr}_2$

The corresponding $\text{ALCI}$ concept $\text{OCL-LiteExpr}^\dagger$ is

$$\neg \text{OCL-LiteExpr}^\dagger_1 \sqcup \text{OCL-LiteExpr}^\dagger_2$$

To further illustrate the mapping, we apply it to some other constraints of our running example. For instance, consider the normalized expression of constraint 1. This constraint is of kind 1, and its subexpressions are respectively of kinds 4, 3, and 2. Applying the corresponding mapping rules we obtain:

$$\text{Person} \sqsubseteq \exists \text{Organizes}.(\text{CriticalEvent} \sqcap \neg \exists \text{SponsoredBy}. \top)$$

As an example of an OCL-Lite expression of kind 6 we have constraint 2. According to the mapping rule, its corresponding $\text{ALCI}$ expression is:

$$\text{Event} \sqsubseteq \neg \exists \text{HeldWith}. \top \sqcup \neg \text{CriticalEvent}$$

Constraint 5 is of kind 5 once normalized. We recursively apply the mapping rules to each part of the disjunction: rules 1 and 3 to the first subexpression, and rules 5, 2, 1 to the second subexpression. The resulting $\text{ALCI}$ expression is:

$$\text{Company} \sqsubseteq \exists \text{SponsoredBy}^{-}. \text{CriticalEvent} \sqcup \exists \text{SponsoredBy}^{-}.(\neg \exists \text{HeldWith}. \top \sqcup \exists \text{HeldWith}. \neg \exists \text{SponsoredBy}. \top)$$

The following theorem states that the mapping from OCL-Lite to $\text{ALCI}$ is correct (we refer to [14] for the proof).

*Theorem 1 (Correctness of the OCL-Lite encoding).* Let $\Gamma$ be a set of OCL-Lite constraints and $\mathcal{K}_\Gamma$ its $\text{ALCI}$ encoding. Then, $\Gamma$ is satisfiable if and only if $\mathcal{K}_\Gamma$ is satisfiable.

Concerning the complexity of reasoning over UML/OCL-Lite schemas we first notice that reasoning just over the UML diagrams as proposed in this paper is an NP-complete problem. Indeed, the UML language we consider here is a sub-language of $\text{ER}_{bool}$ which was proved to be NP-complete in [3], and the very same complexity proof applies to the UML language we use.

*Theorem 2 (Complexity of UML/OCL-Lite).* Checking the satisfiability of UML/OCL-Lite conceptual schemas is an $\text{ExpTime}$-complete problem.

The upper bound follows from the fact that the $\text{ALCI}$ encoding is correct, and that reasoning in $\text{ALCI}$ is in $\text{ExpTime}$. The lower bound is established by reducing satisfiability of $\text{ALC}$ TBoxes, which is known to be $\text{ExpTime}$-complete, to satisfiability of OCL-Lite constraints (see [14] for the proof).

References


