Contents lists available at SciVerse ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

OCL-Lite: Finite reasoning on UML/OCL conceptual schemas

Anna Queralt [a,*], Alessandro Artale [b], Diego Calvanese [b], Ernest Teniente [a]

[a] *Dept. of Service and Information System Engineering, Universitat Politècnica de Catalunya-BarcelonaTech*
[b] *KRDB Research Centre, Free University of Bozen-Bolzano, Italy*

ARTICLE INFO

Article history:
Received 10 August 2010
Received in revised form 16 September 2011
Accepted 16 September 2011
Available online 28 September 2011

Keywords:
OCL constraints
CASE tools + UML
Reasoning
Description Logics

ABSTRACT

To ensure the quality of an information system we must guarantee the correctness of the conceptual schema that represents the knowledge about its domain. The high expressivity of UML schemas annotated with textual OCL constraints enforces the need for automated reasoning techniques. These techniques should be both terminating and complete to be effectively used in practice. In this paper we identify an expressive fragment of the OCL language that ensures these properties. In this way, we overcome the limitations of current techniques when reasoning on such a fragment. As a consequence, we also have that Description Logics can be appropriately used to reason on UML conceptual schemas with arbitrary OCL constraints. We also show how current tools based on different approaches can be used to reason on conceptual schemas enriched with (a decidable fragment of) OCL constraints.

1. Introduction

A conceptual schema consists of a taxonomy of classes together with their attributes, a taxonomy of associations among classes, and a set of integrity constraints over the state of the domain, which define conditions that each instance of the schema must satisfy [32]. These constraints may have a graphical representation or can be defined by means of a particular general-purpose language.

The Unified Modeling Language (UML) [33] 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 OCL [34].

Due to the high expressiveness of the combination of UML and OCL, manually checking the correctness of a conceptual schema becomes a very difficult task, especially when the set of constraints is large. For this reason, it is essential to support the designer with automated reasoning when developing a conceptual schema. Several approaches have been proposed for this purpose [7,35,39,8,24,36,1,37].

The need for reasoning is illustrated by the following example. Consider the UML class diagram in Fig. 1. It specifies a conceptual schema representing events, which are organized and audited by persons. Some events are critical and, in this case, they have at least one responsible. An event can have several sponsors, and can be held with other events. For the sake of simplicity, we have omitted the attributes in the schema, since they do not affect the results of the reasoning we perform.

The UML class diagram is annotated with a set of textual constraints, expressed in OCL and shown in Fig. 2, which provide additional semantics. Constraint 1 states that each person must organize at least an event that is critical and that does not have any sponsor. Constraint 2 ensures that a critical event has at least an inspector. Constraint 3 states that critical events cannot be held with other events. Constraint 4 guarantees that a person cannot organize an event that does not have an inspector. Constraint 5 ensures that all the events audited by a person must have a sponsor. Constraint 6 states that the responsible of a

* Corresponding author at: Dept. of Service and Information System Engineering, Universitat Politècnica de Catalunya, BarcelonaTech, Edifici Omega, c/ Jordi Girona 1, E-08034 Barcelona, Spain. Tel.: +34 934 137 878.
E-mail address: aqueralt@essi.upc.edu (A. Queralt).

0169-023X/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2011.09.004

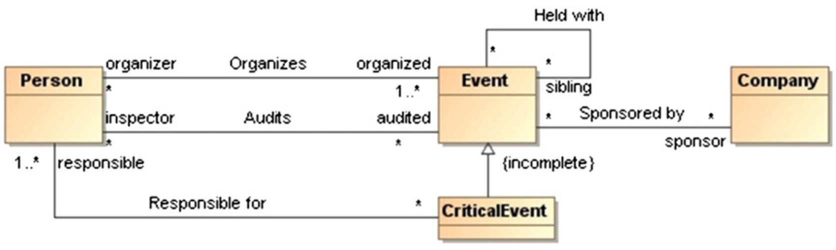


Fig. 1. UML class diagram.

critical event must organize some event. Constraint 7 states that the events that have some organizer must also have a sponsor. Finally, Constraint 8 ensures that each sponsor must be related to some critical event, or it must sponsor at least an event that either does not have any siblings, or it has a sibling without a sponsor.

The conceptual schema in Figs. 1 and 2 is syntactically correct according to the UML and OCL syntax rules. However, this does not ensure that it is semantically correct. In particular, Constraints 1, 2, and 5 are in contradiction, since according to Constraint 1 there must be some critical event that does not have a sponsor, but this is impossible because all critical events must be audited by somebody (Constraint 2), and all the events that are audited must have a sponsor (Constraint 5). Thus, it is impossible to have a valid instance of `Person` that satisfies Constraint 1 and, since each `CriticalEvent` needs a responsible (cardinality constraint) and an inspector (Constraint 2), neither this class can be populated.

This means that the schema is incorrect, and should be fixed. For instance, assume that Constraint 1 is replaced by the following one:

```
context Person inv: organized->exists(oclIsTypeOf(CriticalEvent))
```

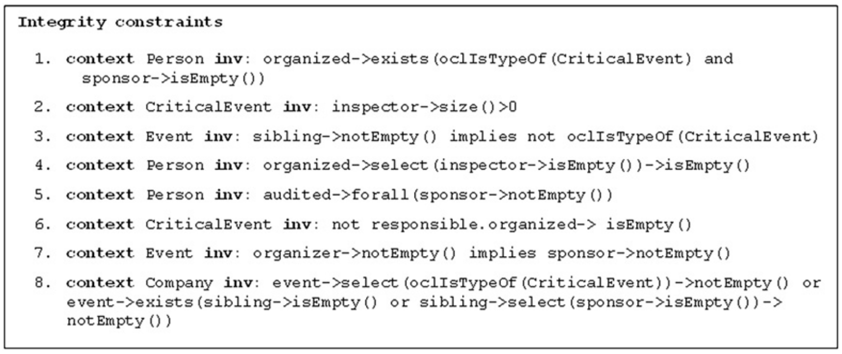
That is, each person must organize at least one critical event. This new constraint allows now all classes and associations to be populated.

However, in conceptual modeling, as well as in databases, assuming that the schema can be instantiated is not enough, because in these communities the possibility of showing *finite* example instantiations is considered very important. In fact, in real applications 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 [9], i.e., the constraints have to admit a finite set of instances. Thus, a schema that admits only infinite instantiations is in practice unsatisfiable from the databases and software engineering point of view [29].

It is well-known that reasoning with OCL integrity constraints in their full generality is undecidable since it amounts to full FOL reasoning. Thus, reasoning with UML conceptual schemas in the presence of OCL constraints has been approached in the following ways:

1. Allowing general constraints (in OCL or other languages) without guaranteeing termination [19,8,35], or guaranteeing termination in some specific cases after analyzing each particular schema [36,37].
2. Allowing general constraints and ensuring termination without guaranteeing completeness of the result [31,39,11,24,10].
3. Ensuring both termination and completeness of finite reasoning by allowing only specific kinds of constraints [29,25,23,40].
4. Ensuring terminating and complete reasoning by disallowing OCL constraints and admitting unrestricted models [16,7,22,3,2].

To our knowledge, none of the existing approaches guarantees complete and terminating reasoning on UML schemas coupled with OCL constraints such as the one in Figs. 1 and 2. Approaches of the first kind do not guarantee termination, that is, a result may not be obtained in some particular cases. On the contrary, the second kind of approaches always terminate, but do not guarantee completeness, that is, they may fail to find existing valid solutions. Approaches of the third kind guarantee both


```
Integrity constraints

 1. context Person inv: organized->exists(oclIsTypeOf(CriticalEvent) and
      sponsor->isEmpty())
 2. context CriticalEvent inv: inspector->size()>0
 3. context Event inv: sibling->notEmpty() implies not oclIsTypeOf(CriticalEvent)
 4. context Person inv: organized->select(inspector->isEmpty())->isEmpty()
 5. context Person inv: audited->forall(sponsor->notEmpty())
 6. context CriticalEvent inv: not responsible.organized-> isEmpty()
 7. context Event inv: organizer->notEmpty() implies sponsor->notEmpty()
 8. context Company inv: event->select(oclIsTypeOf(CriticalEvent))->notEmpty() or
    event->exists(sibling->isEmpty() or sibling->select(sponsor->isEmpty())->
    notEmpty())
```

Fig. 2. OCL constraints for the class diagram in Fig. 1.

completeness and termination but do not allow arbitrary constraints as the ones in Fig. 2. Finally, the last approaches are based on a Description Logic (DL) encoding of a UML schema. They use well known reasoning procedures developed in the DL community to check the satisfiability of a conceptual model. Methods based on DL encodings do not consider OCL constraints and they usually check unrestricted satisfiability (i.e., they allow for infinite instantiations of a schema).

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. To preserve the FMP, the UML class diagram cannot include participation constraints specifying *at-most* constraints, although constructs such as attributes, hierarchies, disjointness, covering, association classes, participation constraints, n-ary associations and key constraints can be handled (see Sections 5.1 and 6 for more details).

The proposed UML/OCL-Lite is the result of identifying a fragment of UML/OCL that can be encoded into the DL $\mathcal{ALCI}$ [13], which has interesting reasoning properties. In particular, $\mathcal{ALCI}$ enjoys the FMP, i.e., every satisfiable set of constraints formalized in $\mathcal{ALCI}$ admits a finite model. Thus, the FMP is also guaranteed for any fragment of OCL that fits into $\mathcal{ALCI}$. The contributions of this paper can be summarized as follows:

- The identification of UML/OCL-Lite, a fragment of both UML and 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 UML/OCL-Lite to the DL $\mathcal{ALCI}$ to prove that the proposed fragment has the same reasoning properties as $\mathcal{ALCI}$. 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 $\mathcal{ALCI}$ we are able to show both the FMP that checking satisfiability in UML/OCL-Lite is an EXPTIME-complete problem.
- We show how current technology can be effectively used to automatically reason on a UML/OCL-Lite schema. In particular, a DL reasoner is used for the first time to check several desirable properties of a conceptual schema coupled with OCL constraints. We also show a semidecision procedure able to perform the same reasoning tasks, which, as expected, always terminates within this fragment of OCL.

The rest of the paper is structured as follows. Section 2 reviews how the problem of reasoning on conceptual schemas with constraints has been addressed in the literature. In Section 3, we provide the syntax rules of OCL-Lite, as well as the semantics of this fragment of the language. Section 4 explains the mapping from a UML/OCL-Lite conceptual schema to $\mathcal{ALCI}$. In Section 6, we show how identifiers, which cannot be expressed in $\mathcal{ALCI}$, can be handled by our approach. In Section 7, we show how our proposal is supported by two existing tools following different approaches. Finally, in Section 8, we explain our conclusions and point out future research directions.

2. Related work

In this section, we review how the problem of reasoning on conceptual schemas has been addressed in the literature. Reasoning on conceptual schemas with OCL integrity constraints in their full generality is undecidable since it amounts to reasoning in FOL. Thus, no reasoning procedures that deal with general OCL constraints can be both correct and terminating. The existing approaches dealing with conceptual modeling and OCL constraints can be divided into those renouncing to decidability, those renouncing to completeness, and those looking for decidable fragments.

Several approaches renounce to decidability in favor of expressiveness, thus they are able to deal with general constraints but may not terminate in some cases. In this group we mention RoZ [19], which considers UML schemas with arbitrary constraints specified in Z instead of OCL, and [8,35] that deal with UML schemas with OCL constraints. A different approach in this group is the framework presented in [36,37], which imposes mild restrictions on the expressiveness of the constraints in order to ensure decidability in a greater number of cases. Based on structural properties of both the schema and the OCL constraints the authors propose a method to check whether the reasoner will terminate.

Other approaches rely on incomplete but terminating procedures in order to deal with general constraints. An important approach of this kind is the combination called UML2Alloy [1], which encodes a subset of UML and OCL into the Alloy language [27]. By using then the Alloy analyzer [31], the authors provide reasoning capabilities for schemas formalized in a logic notation. Given a schema, Alloy tries to find possible models of a fixed finite size. The procedure is incomplete, since failure to find such a fixed size model does not necessarily mean that the schema is unsatisfiable. The work in [10] translates an UML/OCL schema into constraint programming and then uses a CSP solver. As in the previous approach, the procedure searches for possible models of a fixed finite size and, thus, the procedure in not complete. The work in [24] presents a procedure that, given an instantiation of the UML/OCL schema (provided as an input by the user), verifies whether such a particular instantiation is a model of the schema. Also in this case the procedure is obviously not complete. Recently, the ability to use a SAT-solver has been incorporated in the approach, so that the user does not need to provide an instantiation [28]. However the search space still needs to be limited by fixing the size of the models to be found by the solver. Another approach dealing with UML schemas is presented in [39], which requires that constraints are specified in B. A different approach is the one in [40], which defines consistency rules between a list of OCL constraint patterns. This approach guarantees termination but is not complete since there are false positive cases, i.e., consistent constraints can be displayed as potentially inconsistent.

Finally, there are approaches that limit the expressiveness by allowing only specific kinds of constraints in order to guarantee both decidability and completeness. These approaches usually disregard OCL constraints and concentrate on a subset of the full UML/EER graphical constraints. The work presented in [29] studies the complexity of satisfiability checking taking into account cardinality constraints, in [15] ISA constraints between classes are also added, while [25] considers the case where identifiers are present. In [23], object-oriented database schemas are considered with cardinality restrictions and constraints expressing the range of attributes.

Finally, we briefly mention those approaches based on encodings into different DLs. The DL encoding presented in [7] shows that reasoning over full EER/UML schemas is an EXPTIME-complete problem. The upper bound is obtained by mapping UML schemas into the $\mathcal{ALCQI}$ logic, while the lower bound derives from reducing the problem of Knowledge Base (KB) satisfiability in $\mathcal{ALC}$ (known to be EXPTIME-complete) to checking the satisfiability of a UML schema. The study presented in [3] shows that better complexity results can be gained by limiting the expressive power of EER/UML schemas disallowing sub-relationships and covering constraints (in particular, NP and NLOGSPACE, respectively). The results are obtained by mapping to so called *DL-Lite* logics. Recently, the approach of encoding a schema into a DL has also been used for inferring knowledge from conceptual schemas of a particular domain [6] (in this case, data-access request scenarios in healthcare systems). The languages used are a fragment of OWL (Web Ontology Language) and SWRL (Semantic Web Rule Language), both of them based on DLs, and with a reasoning complexity that, in this case, is PSpace-complete. Note that, the approaches based on a DL encoding, in the general case, do not guarantee the FMP.

The main contribution of our approach is to deal with expressive conceptual schemas, specified using both UML and OCL, guaranteeing finite reasoning with a complete procedure. Similarly to the approaches based on a DL encoding, we encode an UML schema using an analogous technique. On the other hand, we extend such encoding by devising an appropriate subset of the full OCL language, i.e., OCL-Lite, and then mapping OCL-Lite to the DL $\mathcal{ALCI}$. To our knowledge, this is the first approach that allows to reason on UML conceptual schemas with OCL constraints using DLs. Furthermore, differently from the works mentioned above, our encoding enjoy the FMP.

3. 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 class diagrams and OCL constraints. For further details on the syntax and semantics of OCL expressions, we refer the reader to [34,41].

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 defined 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. Sometimes, the role name for an association end is omitted, and then it is assumed that the role played by the participant is the name of the corresponding class.

For instance, referring to the UML class diagram in Fig. 1, the association `SponsoredBy` has two association ends. One of them corresponds to the class `Company`, which plays the role of `sponsor`, and the other one corresponds to class `Event`, which plays the role of `event` in this association.

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. The reserved word `self` may be optionally used within *OCLExpr* to refer to an arbitrary instance of the contextual class.

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. If no role name is specified in the class diagram for a given association end, the name of the destination class is used in the navigation. The first role name in the path (optionally preceded by `self`) refers to an association end that is accessible from the contextual class, i.e., an association end belonging to an association to which the contextual class participates. The rest of the navigation path is defined analogously.

For instance, referring again to Fig. 1, and assuming that the context is `Company`, the following expression results in the events sponsored by an arbitrary instance of `Company`, by means of the association `SponsoredBy`:

`self.event` or equivalently `event`

The following expression returns the persons that organize an arbitrary instance of `Event`, by means of the association `Organizes`, assuming that the context is `Event`:

`self.organizer` or equivalently `organizer`

And the following one gives the events that are organized by a person, assuming that the context is `Person`:

`self.organized` or equivalently `organized`

3.1. 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 proposed syntax rules, an *OCL-Lite expression OCL-LiteExpr* is defined recursively, since OCL expressions can be combined to obtain new ones.

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()

Intuitively, *OCL-LiteExpr* allows one to construct a Boolean OCL-Lite expression, which can correspond to the whole constraint, or can be the condition 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 the application of 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, 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;

Operation oclIsTypeOf(*C*) applies only to a single object *o*. Its intuitive meaning is the following:

- *o*.oclIsTypeOf(*C*): returns true iff *o* is an instance of the class *C*;

Optionally, in those OCL operations that apply to collections, one can give a name to refer to an arbitrary element of the collection to which the operation is applied. For instance, for the operation select, this is done by means of the expression

$$col{-}{>}select(o \mid OCL{-}LiteExpr{-}with{-}o)$$

where *o* is called the *iterator*, and is a reference to an object from the collection *col*. When the select is evaluated, *o* iterates over the collection and *OCL-LiteExpr-with-o* is evaluated for each *o*. For instance, examples of valid OCL-Lite expressions are:

- r_1->exists(r_2.r_3->size()>0)
  or, equivalently:
  self.r_1->exists(o|o.r_2.r_3->size()>0),
- r_1.oclAsType(C_2).r_2->select(r_3->isEmpty())->forall(oclIsTypeOf(C_4))
  or, equivalently:
  self.r_1.oclAsType(C_2).r_2->select(o|o.r_3->isEmpty())->forall(o|o.oclIsTypeOf(C_4))
- r_1->select(r_2->isEmpty())->select(r_3.r_4->exists(r_5->notEmpty()))
  ->forall(oclIsTypeOf(C) and r_6->size()=0)
  or, equivalently:
  self.r_1->select(o|o.r_2->isEmpty())->select(o|o.r_3.r_4->exists(o_2|o_2.r_5->notEmpty()))->
  forall(o|o.oclIsTypeOf(C) and o.r_6->size()=0)

Since both the variable self and iterator variables are optional in OCL expressions, we will omit them in the rest of the paper for the sake of simplicity. All constraints in Fig. 2 are examples of well-formed OCL-Lite expressions.

3.2. OCL-Lite normal form

OCL-Lite operations, except for `oclIsTypeOf` and `oclAsType`, can be expressed only in terms of `select`, `isEmpty`, and `notEmpty`. Thus, we first rewrite each OCL-Lite expression as an equivalent *normalized* one, which is expressed in terms of these operations only. Table 1 shows the OCL-Lite operations and gives their equivalent normalized expressions. 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 the expression `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 Fig. 2 that have to be normalized are 1, 2, 5, 6, and 8. The resulting expressions we get after applying the rules in Table 1 are:

- Constraint 1. We apply rule a) to the original constraint and we get the normalized expression:

```
context Person inv:
   organized->select(oclIsTypeOf(CriticalEvent) and sponsor->isEmpty())->notEmpty()
```

- Constraint 2. We apply rule d) and we get the normalized expression:

```
context CriticalEvent inv: inspector->notEmpty()
```

- Constraint 5. 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 6. We apply rule f) and we get:

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

- Constraint 8. We apply rule a) and we get:

```
context Company 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. In particular, the final result is always a combination of expressions basically consisting of an optional `select` operation followed either by `isEmpty` or `notEmpty`. Also, the expression may include the operation `oclIsTypeOf`. Importantly, if either the original expression or an intermediate one contains a sequence of `select` operations, they will be collapsed in a single one when applying the normalization rules.

3.3. OCL-Lite semantics

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-LiteExpr* into a first order logic (FOL) formula $\mathit{OCL\text{-}LiteExpr}^f(x)$ with one free variable. Other approaches specify the semantics of OCL expressions using first-order terms instead of formulas [5]. However, as also argued in [5], using formulas is preferable when dealing with sets, as in our case.

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

`context` *C* `inv`: *OCL-LiteExpr*

Table 1
Normalization of OCL-Lite expressions.

	Original expression	Normalized expression
a)	*col*`->exists(cond)`	*col*`->select(cond)->notEmpty()`
b)	*col*`->forall(cond)`	*col*`->select(not cond)->isEmpty()`
c)	*col*`->select(`$\mathtt{cond}_1$`)->select(`$\mathtt{cond}_2$`)`	*col*`->select(`$\mathtt{cond}_1$` and `$\mathtt{cond}_2$`)`
d)	*col*`->size()>0`	*col*`->notEmpty()`
e)	*col*`->size()=0`	*col*`->isEmpty()`
f)	`not` *col*`->isEmpty()`	*col*`->notEmpty()`
g)	`not` *col*`->notEmpty()`	*col*`->isEmpty()`

Its interpretation is:

$$\forall x\Big(C(x) \rightarrow \textit{OCL-LiteExpr}^f(x)\Big)$$

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 $p_n \ldots p_1$ in an OCL-Lite expression, where each p_i is either a role name or `oclAsType(`C_i`)`.r_i. To formalize a (binary) association A_i, we introduce a binary predicate, A_i, whose first argument represents an instance of $dom(A_i)$ (the domain of A_i) and whose second argument represents an instance of $range(A_i)$ (the range of A_i). To fix a semantics for role names we conform to the UML convention about role names [38], i.e., a role name attached to an association end labeled with a class C is used to navigate from one object to a another one belonging to the class C. Thus, in the following, $p_i^f(x,y) = A_i(x,y)$ when p_i is a role name attached to the $range(A_i)$-end of A_i, and, viceversa, $p_i^f(x,y) = A_i(y,x)$ when p_i is a role name attached to the $dom(A_i)$-end of A_i. Similarly, $p_i^f(x,y) = C_i(x) \wedge A_i(x,y)$, when $p_i =$ `oclAsType(`C_i`)`.r_i and `r`$_i$ is a role name attached to the $range(A_i)$-end of A_i, while $p_i^f(x,y) = C_i(x) \wedge A_i(y,x)$, when $p_i =$ `oclAsType(`C_i`)`.r_i and r_i is a role name attached to the $dom(A_i)$-end of A_i. Note that, some of the expressions (and their interpretations) are defined recursively, since OCL-Lite expressions can be combined to obtain new expressions.

1. *OCL-LiteExpr* $= p_n \ldots p_1$`->select(`*OCL-LiteExpr*$_0$`)->notEmpty()`
   The semantics of this expression is

   $$\textit{OCL-LiteExpr}^f(x) = \exists x_n \cdots \exists x_1 \left(p_n^f(x,x_n) \wedge p_{n-1}^f(x_n,x_{n-1}) \wedge \cdots \wedge p_1^f(x_2,x_1) \wedge \textit{OCL-LiteExpr}_0^f(x_1)\right)$$

   Importantly, a particular case of this kind of expression is when no `select` operation is applied on the objects obtained from the navigation, which corresponds to the expression

   $$\textit{OCL-LiteExpr} = p_n \ldots p_1 \text{->}\texttt{notEmpty}()$$

   In this case, the semantics is

   $$\textit{OCL-LiteExpr}^f(x) = \exists x_n \cdots \exists x_1 \left(p_n^f(x,x_n) \wedge p_{n-1}^f(x_n,x_{n-1}) \wedge \cdots \wedge p_1^f(x_2,x_1)\right)$$

2. *OCL-LiteExpr* $= p_n \ldots p_1$`->select(`*OCL-LiteExpr*$_0$`)->isEmpty()`
   The semantics of this expression is

   $$\textit{OCL-LiteExpr}^f(x) = \forall x_n \cdots \forall x_1 \left(\neg p_n^f(x,x_n) \vee \cdots \vee \neg p_1^f(x_2,x_1) \vee \neg \textit{OCL-LiteExpr}_0^f(x_1)\right)$$

   Again, we have a particular case of this kind of expression in the absence of `select`:

   $$\textit{OCL-LiteExpr} = p_n \ldots p\text{->}\texttt{isEmpty()}$$

   And then, the semantics of the expression is

   $$\textit{OCL-LiteExpr}^f(x) = \forall x_n \cdots \forall x_1 \left(\neg p_n^f(x,x_n) \vee \cdots \vee \neg p_1^f(x_2,x_1)\right)$$

3. *OCL-LiteExpr* $=$ [`not`] `oclIsTypeOf(`C`)`
   where the brackets denote optionality. The semantics of the expression is

   $$\textit{OCL-LiteExpr}^f(x) = [\neg]C(x)$$

4. *OCL-LiteExpr* $=$ *OCL-LiteExpr*$_1$ `and` *OCL-LiteExpr*$_2$
   The semantics of this expression is

   $$\textit{OCL-LiteExpr}^f(x) = \textit{OCL-LiteExpr}_1^f(x) \wedge \textit{OCL-LiteExpr}_2^f(x)$$

5. *OCL-LiteExpr* $=$ *OCL-LiteExpr*$_1$ `or` *OCL-LiteExpr*$_2$
   The semantics of this expression is

   $$\textit{OCL-LiteExpr}^f(x) = \textit{OCL-LiteExpr}_1^f(x) \vee \textit{OCL-LiteExpr}_2^f(x)$$

6. *OCL-LiteExpr* = *OCL-LiteExpr*$_1$ `implies` *OCL-LiteExpr*$_2$
   The semantics of this expression is

$$OCL\text{-}LiteExpr^f(x) = OCL\text{-}LiteExpr_1^f(x) \rightarrow OCL\text{-}LiteExpr_2^f(x)$$

In our running example, the semantics of Constraints 1 and 2 is the following:

- The OCL-Lite normal form of Constraint 1 is

```
context Person inv:
   organized->select(oclIsTypeOf(CriticalEvent) and sponsor->isEmpty())->notEmpty()
```

  and its semantics is

$$\forall x(Person(x) \rightarrow \exists y_1(Organizes(x, y_1) \wedge CriticalEvent(y_1) \wedge \forall y_2 \neg SponsoredBy(y_1, y_2)))$$

- The OCL-Lite normal form of Constraint 2 is

```
context CriticalEvent inv: inspector->notEmpty()
```

  and its semantics is

$$\forall x(CriticalEvent(x) \rightarrow \exists y Audits(y, x))$$

Definition 1. Satisfiability of OCL-Lite constraints

Let Γ be a set of OCL-Lite constraints and Γ^f the resulting FOL theory. Then, Γ is said to be satisfiable if there exists a first order interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ that satisfies Γ^f. $\mathcal{I}$ is called a model of Γ. □

4. The Description Logic $\mathcal{ALCI}$

One of the distinguishing features of DLs is that they admit terminating reasoning procedures that are sound and complete with respect to the semantics. Additionally, $\mathcal{ALCI}$ [13] has the FMP [17,15], which means that every satisfiable formula of the logic admits a *finite model*, i.e., a model with a finite domain. Intuitively, this means that it is impossible to define with this language a set of constraints that necessarily needs an infinite number of instances to be satisfied. This guarantees that if a schema is found to be satisfiable, then it is finitely satisfiable, which is the notion of satisfiability that is assumed in the software engineering community.

Importantly, finite model reasoning has been studied for more expressive DLs that do not enjoy the FMP, such as $\mathcal{ALCQI}$ [4,12,30]. This means that it is still possible to determine whether a schema specified in this language is finitely satisfiable, but no tools or prototypes that perform this check have ever been implemented. Additionally, our aim is not to stick to any particular reasoning approach, but to provide a specification language that is both familiar to the conceptual modeling community and that can be handled by currently existing tools in this area, not necessarily by techniques based on DLs.

Thus, we choose $\mathcal{ALCI}$ as our target language since, due to its FMP, finiteness of the domain can always be assumed. Hence, a procedure that searches for instantiations that satisfy a certain property, as most existing approaches in conceptual modeling do, can be used with the guarantee of termination.

In the following, we present the syntax and semantics of $\mathcal{ALCI}$, which is an expressive DL in which knowledge is represented by means of *concepts* (unary predicates) and *roles* (binary predicates). Concepts, denoted by *D*, and roles, denoted by *R*, are formed according to the following syntax rules:

$$\begin{array}{lll} D & ::= & \top \mid C \mid \neg D \mid D \sqcap D' \mid D \sqcup D' \mid \exists R.D \mid \forall R.D \\ R & ::= & A \mid A^- \end{array}$$

where *C* denotes an *atomic concept*, and *A* an *atomic role*, i.e., simply a concept or role symbol.

An $\mathcal{ALCI}$ knowledge base is constituted by a finite set of inclusion assertions of the form $D_1 \sqsubseteq D_2$, where D_1 and D_2 are arbitrary concept expressions.[1]

[1] Here we deal only with knowledge at the intentional level, and do not consider extensional knowledge, i.e., knowledge about individual objects. Hence, we identify a DL knowledge base with its intentional component, usually called TBox [4].

The semantics of $\mathcal{ALCI}$ is specified through the notion of interpretation. An *interpretation* $\mathcal{I}=(\Delta^{\mathcal{I}},\cdot^{\mathcal{I}})$ of an $\mathcal{ALCI}$ knowledge base is constituted by an interpretation domain $\Delta^{\mathcal{I}}$ and an interpretation function $\cdot^{\mathcal{I}}$ that assigns to each concept D a subset $D^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$ and to each role R a subset $R^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}\times\Delta^{\mathcal{I}}$, such that the following conditions are satisfied:

$$
\begin{array}{rcl@{\qquad}rcl}
\top & = & \Delta^{\mathcal{I}} & (\exists R.D)^{\mathcal{I}} & = & \left\{o\in\Delta^{\mathcal{I}} \middle| \exists o'.\left(o,o'\right)\in R^{\mathcal{I}}\wedge o'\in D^{\mathcal{I}}\right\} \\
C^{\mathcal{I}} & \subseteq & \Delta^{\mathcal{I}} & (\forall R.D)^{\mathcal{I}} & = & \left\{o\in\Delta^{\mathcal{I}} \middle| \forall o'.\left(o,o'\right)\in R^{\mathcal{I}}\rightarrow o'\in D^{\mathcal{I}}\right\} \\
\neg D^{\mathcal{I}} & = & \Delta^{\mathcal{I}}\setminus D^{\mathcal{I}} & A^{\mathcal{I}} & \subseteq & \Delta^{\mathcal{I}}\times\Delta^{\mathcal{I}} \\
(D_1\sqcap D_2)^{\mathcal{I}} & = & D_1^{\mathcal{I}}\cap D_2^{\mathcal{I}} & (A^{-})^{\mathcal{I}} & = & \left\{\left(o,o'\right)\in\Delta^{\mathcal{I}}\times\Delta^{\mathcal{I}} \middle| \left(o',o\right)\in A^{\mathcal{I}}\right\} \\
(D_1\sqcup D_2)^{\mathcal{I}} & = & D_1^{\mathcal{I}}\cup D_2^{\mathcal{I}} & & &
\end{array}
$$

An interpretation $\mathcal{I}$ *satisfies* an inclusion assertion $D_1\sqsubseteq D_2$ if $D_1^{\mathcal{I}}\subseteq D_2^{\mathcal{I}}$. An interpretation that satisfies all assertions in a knowledge base $\mathcal{K}$ is called a *model* of $\mathcal{K}$. A knowledge base $\mathcal{K}$ is *satisfiable* if there exists a model of $\mathcal{K}$. A concept D is *satisfiable* w.r.t $\mathcal{K}$ if there is a model $\mathcal{I}$ of $\mathcal{K}$ such that $D^{\mathcal{I}}$ is nonempty.

Intuitively, the knowledge that can be expressed in $\mathcal{ALCI}$ is encoded in terms of assertions involving concepts (D) and roles (R). Roles can be seen as binary associations in UML, and the inverse role, denoted A^{-}, corresponds to the inverse of an association. Each (inverse) role has a *domain* and a *range*. For instance, the association *Organizes* in our example can be formalized in $\mathcal{ALCI}$ by means of a role Organizes that denotes the pairs constituted by a person and an event such that the person (domain) organizes the event (range). Then, Organizes^{-} relates the event (domain) to the persons (range) organizing it.

Regarding concepts, they may be atomic (C), which allows them to represent the entities in a conceptual schema, or arbitrary (D), which are used in the specification of constraints. In particular, one can specify the negation of an arbitrary concept, the union or intersection of two arbitrary concepts, or build more complex concepts involving roles by means of *quantified role restrictions* as follows. A concept $\exists R.D$ denotes the instances of the domain of R that are related to some instance of the concept D, and $\forall R.D$ denotes the objects that are related by R only to instances of the concept D. Notice that this includes those objects that are not related by R to any object of the domain. For instance, $\exists$ Organizes.CriticalEvent denotes the individuals that organize some critical event, and $\forall$Organizes^{-}.Person denotes the events that are organized only by persons. As will be seen, more complex expressions can be build by using arbitrary concepts instead of atomic ones when specifying $\mathcal{ALCI}$ concepts.

5. Encoding UML/OCL-Lite in $\mathcal{ALCI}$

In this section we show that the proposed fragment UML/OCL-Lite can be encoded in $\mathcal{ALCI}$ and, thus, finite reasoning is guaranteed for every schema expressed in this modeling language. We first present the fragment of UML we are interested in together with its encoding, and then we provide an encoding for the OCL-Lite fragment, too.

5.1. Encoding UML in $\mathcal{ALCI}$

In the following, we give an overview of how to encode a fragment of UML class diagrams in $\mathcal{ALCI}$, based on the encoding in $\mathcal{ALCQI}$ proposed in [7]. Since $\mathcal{ALCQI}$ is an extension of $\mathcal{ALCI}$ that does not have the FMP (i.e., a schema specified in $\mathcal{ALCQI}$ might be satisfiable only by an infinite number of instances), we focus on a fragment of UML that can be encoded into $\mathcal{ALCI}$. As a consequence, there are some constructs of UML class diagrams that cannot be encoded in this language. 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). Moreover, due to readability issues, in this section we do not consider UML constructors such as *identification constraints*, *association classes*, or *n-ary associations*. However, as will be clarified in Section 6, they can also be captured in the $\mathcal{ALCI}$ encoding.

Given an UML class diagram, we encode each class C into an atomic concept C, each (binary) association A into an atomic role A, each attribute a into an atomic role a, and each type T into an atomic concept T. To express that types are disjoint both between themselves and from classes, the following disjointness assertions are enforced:

$$
\begin{array}{lll}
T_1 & \sqsubseteq \neg T_2, & \text{for every pair of distinct types } T_1 \text{ and } T_2, \text{ and} \\
T & \sqsubseteq \neg C, & \text{for every type } T \text{ and class } C.
\end{array}
$$

An UML schema is encoded as a set of $\mathcal{ALCI}$ inclusion assertions (i.e., an $\mathcal{ALCI}$ knowledge base) as described in the following.

- A *hierarchy* constraint expressing a generalization of a class C_1 into a super-class C_2 is encoded as

  $$C_1 \sqsubseteq C_2$$

- A *disjointness* constraint among classes $C_1,\ldots,C_n$ that are sub-classes of a class C, is encoded as

  $$C_i \sqsubseteq \sqcap_{j=i+1}^{n} \neg C_j, \quad \text{with } 1 \leq i \leq n-1$$

- A *covering* constraint involving classes C, $C_1,\ldots,C_n$ is encoded as

  $$C \sqsubseteq \sqcup_{j=1}^{n} C_j$$

- A *typing* constraint for an attribute a of a class C is encoded as

  $$C \sqsubseteq \forall a.T$$

  where T is a class representing the type of the attribute.

Each *association* A between classes C_1 (domain) and C_2 (range) is encoded by an atomic role A, together with the following inclusion assertion to specify the domain and range of A:

$$\top \sqsubseteq \forall A^{-}.C_1 \sqcap \forall A.C_2$$

Both for attributes and for associations, the cardinalities (multiplicities, for attribute) that can be encoded in $\mathcal{ALCI}$ are * and `1..*`. No inclusion assertion is needed to encode the former, since it corresponds to the absence of a constraint. The inclusion assertion that encodes the *multiplicity* `1..*` of an attribute a of C is:

$$C \sqsubseteq \exists a.\top$$

A *cardinality* `1..*` attached to the association end corresponding to C_1 in the association A is encoded as[2]:

$$C_2 \sqsubseteq \exists A^{-}.\top$$

and, viceversa, the following inclusion assertion captures the cardinality `1..*` attached to the association end corresponding to C_2:

$$C_1 \sqsubseteq \exists A.\top$$

According to the above rules, the class diagram in Fig. 1 is encoded in $\mathcal{ALCI}$ as follows:

1. $\top \sqsubseteq \forall \mathsf{Organizes.Event} \sqcap \forall \mathsf{Organizes}^{-}.\mathsf{Person}$
2. $\top \sqsubseteq \forall \mathsf{Audits.Event} \sqcap \forall \mathsf{Audits}^{-}.\mathsf{Person}$
3. $\top \sqsubseteq \forall \mathsf{ResponsibleFor.CriticalEvent} \sqcap \forall \mathsf{ResponsibleFor}^{-}.\mathsf{Person}$
4. $\top \sqsubseteq \forall \mathsf{HeldWith.Event} \sqcap \forall \mathsf{HeldWith}^{-}.\mathsf{Event}$
5. $\top \sqsubseteq \forall \mathsf{SponsoredBy.Company} \sqcap \forall \mathsf{SponsoredBy}^{-}.\mathsf{Event}$
6. $\mathsf{Person} \sqsubseteq \exists \mathsf{Organizes}.\top$
7. $\mathsf{CriticalEvent} \sqsubseteq \exists \mathsf{ResponsibleFor}^{-}.\top$
8. $\mathsf{CriticalEvent} \sqsubseteq \mathsf{Event}$

Inclusion assertions 1 to 5 specify the typing (i.e., domain and range) of the associations in the schema. For instance, the first one states that the range of *Organizes* is *Event* and its domain is *Person*. Inclusion assertions 6 and 7 correspond to the cardinality constraints `1..*` specified in the associations *Organizes* and *ResponsibleFor*. For example, the first one says that each *Person* must *Organize(s)* at least one *Event*. Inclusion assertion 8 captures that *CriticalEvent* is a subclass of *Event*. Note that role names are not translated as far as the UML schema contains only binary associations. They will play a role when n-ary associations are captured via reification (see Section 6 for more details).

The encoding, starting from an UML class diagram Σ, generates an $\mathcal{ALCI}$ knowledge $\mathcal{K}_\Sigma$. The encoding is correct, i.e., it preserves *satisfiability* (Σ is satisfiable if and only if $\mathcal{K}_\Sigma$ is satisfiable) since it applies similar rules as the proven correct encoding presented in [7]. Note that, the domain of the interpretation of such an $\mathcal{ALCI}$ knowledge base corresponds to the set of objects that instantiate the UML class diagram. Since there are neither association classes nor n-ary associations, we do not use reification, and thus objects instantiating classes correspond to instances of the corresponding DL concepts, and pairs

[2] We conform to the conventional reading of cardinality constraints in UML [38].

of objects instantiating (binary) associations correspond to instances of the corresponding DL roles. Hence, a model of the knowledge base $\mathcal{K}_\Sigma$ can be viewed as an instantiation of the UML class diagram Σ.

5.2. Encoding OCL-Lite constraints in $\mathcal{ALCI}$

Taking into account the $\mathcal{ALCI}$ encoding of an UML class diagram explained above, in the following we provide a mapping to translate OCL-Lite constraints into $\mathcal{ALCI}$.

An OCL-Lite constraint, which has the general form

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

is encoded as the following $\mathcal{ALCI}$ inclusion assertion:

$$C \sqsubseteq \mathit{OCL\text{-}LiteExpr}^\dagger$$

where $\cdot^\dagger$ is a mapping function that assigns to each OCL-Lite expression its corresponding $\mathcal{ALCI}$ encoding. This inclusion assertion states that the set of instances of the concept C (encoding the context class C) is a subset of the instances of the concept that encodes the *OCL-LiteExpr*. In other words, according to the semantics of OCL constraints, each instance of C must satisfy *OCL-LiteExpr*.

In the following, we illustrate the encoding of OCL-Lite expressions in $\mathcal{ALCI}$. We consider OCL-Lite expressions in their normal form, as provided in Section 3.2, and define *OCL-LiteExpr*† by induction on the structure of *OCL-LiteExpr*.

1. $\mathit{OCL\text{-}LiteExpr} = p_n \ldots p_1$`->select(`$\mathit{OCL\text{-}LiteExpr}_0$`)->notEmpty()`

   We define the $\mathcal{ALCI}$ concept *OCL-LiteExpr*† 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 $\mathit{OCL\text{-}LiteExpr}^\dagger = \mathit{OCL\text{-}LiteExpr}_0^\dagger$.

   For the inductive case, let $\mathit{OCL\text{-}LiteExpr}_n = p_n \ldots p_1$`->select(`$\mathit{OCL\text{-}LiteExpr}_0$`)->notEmpty()`, let p_{n+1} be an additional path item, and let $\mathit{OCL\text{-}LiteExpr}_{n+1} = p_{n+1}.\mathit{OCL\text{-}LiteExpr}_n$. Then $\mathit{OCL\text{-}LiteExpr}_{n+1}^\dagger = p_{n+1}^\dagger.(\mathit{OCL\text{-}LiteExpr}_n^\dagger)$, where $p_{n+1}^\dagger$ (for the various cases of p_{n+1}, cf. OCL syntax in Section 3.1) is an abbreviation[3] defined as follows (r denotes a role name of an association A, and $dom(A)$ and $range(A)$ denote respectively the domain and range of A):

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

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

   Note that the $\mathcal{ALCI}$ concept corresponding to *OCL-LiteExpr* has the form

$$p_n^\dagger.\left(p_{n-1}^\dagger\left(\cdots\left(p_n^\dagger.\left(\mathit{OCL\text{-}LiteExpr}_0^\dagger\right)\right)\cdots\right)\right)$$

   Intuitively, this concept represents the fact that *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 $\mathit{OCL\text{-}LiteExpr}_0$. In the particular case when there is no `select` operation, i.e., the OCL expression has the form $p_n \ldots p_1$`->notEmpty()`, then $\mathit{OCL\text{-}LiteExpr}_0^\dagger = \top$, and the constraint is encoded as

$$p_n^\dagger.\left(p_{n-1}^\dagger.\left(\cdots\left(p_1^\dagger.\top\right)\cdots\right)\right)$$

   that is, no condition is imposed on those instances reachable through the path.

   For example, an expression in our running example that follows this pattern is the one in the body of Constraint 6. Once normalized, the constraint has the form

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

   Thus, the $\mathcal{ALCI}$ assertion that encodes this constraint is:

   $\mathsf{CriticalEvent} \sqsubseteq \exists\mathsf{ResponsibleFor}^-.(\exists\mathsf{Organizes}.\top)$

   Note that, the $\mathcal{ALCI}$ encoding of the OCL-Lite expression `responsible.organized->notEmpty()` is $\exists\mathsf{ResponsibleFor}^-.(\exists\mathsf{Organizes}.\top)$. The first DL role, $\mathsf{ResponsibleFor}^-$, is inverted since the association `ResponsibleFor` has domain `Person` and range `CriticalEvent`, and the role name `responsible` is attached to `Person`. Thus, `responsible`$^\dagger = (\exists\mathsf{ResponsibleFor}^-)$. The next role name in the OCL-Lite expression is `organized`, which is attached to `Event`, the range of the association `Organizes`.

[3] Note that $p^\dagger$ is not a valid DL expression.

Thus, `organized`$^\dagger = \exists$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. $\mathit{OCL\text{-}LiteExpr} = p^n \dots p^1$`->select(`$\mathit{OCL\text{-}LiteExpr}^0$`)->isEmpty()`
Similarly to the previous case, we define $\mathit{OCL\text{-}LiteExpr}$= by induction on n. For the base case of $n = 0$, we set $\mathit{OCL\text{-}LiteExpr}^\dagger = \neg \mathit{OCL\text{-}LiteExpr}_0^\dagger$. For the inductive case, let $\mathit{OCL\text{-}LiteExpr}_n = p_n \dots p_1$`->select(`$\mathit{OCL\text{-}LiteExpr}_0$`)->isEmpty()`, let p_{n+1} be an additional path item, and let $\mathit{OCL\text{-}LiteExpr}_{n+1} = p_{n+1}\,.\,\mathit{OCL\text{-}LiteExpr}_2$. Then $\mathit{OCL\text{-}LiteExpr}_{n+1}^\dagger = \neg p_n^\dagger.(\neg \mathit{OCL\text{-}LiteExpr}_n^\dagger)$, Considering that $\neg\neg C$ is equivalent to C, the $\mathcal{ALCI}$ concept corresponding to $\mathit{OCL\text{-}LiteExpr}$ has the form

$$\neg\left(p_n^\dagger.\left(p_{n-1}^\dagger.\left(\cdots\left(p_1^\dagger.\left(\mathit{OCL\text{-}LiteExpr}_0^\dagger\right)\right)\cdots\right)\right)\right)$$

Intuitively, this concept represents the fact that $\mathit{OCL\text{-}LiteExpr}$ evaluates to true, for a given instance o, if o is not related through the path $p_n \dots p_1$ to any object that satisfies the condition specified by $\mathit{OCL\text{-}LiteExpr}_0$. As in the previous case, if there is no `select` operation in the OCL expression, i.e., the OCL expression has the form $p_n \dots p_1$`->isEmpty()`, then $\mathit{OCL\text{-}LiteExpr}_0^\dagger = \top$, and the constraint is encoded as

$$\neg\left(p_n^\dagger.\left(p_{n-1}^\dagger.\left(\cdots\left(p_1^\dagger.\top\right)\cdots\right)\right)\right)$$

As an example, let us consider Constraint 5 in its normal form:

```
context Person inv: audited->select (sponsor->isEmpty())->isEmpty()
```

The overall OCL-Lite expression is encoded in $\mathcal{ALCI}$ as $\neg(\exists\text{Audits}.(\neg\exists\text{SponsoredBy}.\top))$, and the $\mathcal{ALCI}$ assertion corresponding to the constraint is:

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

3. $\mathit{OCL\text{-}LiteExpr} =$ `oclIsTypeOf(`C`)`, and $\mathit{OCL\text{-}LiteExpr} =$ `not oclIsTypeOf(`C`)`
The $\mathcal{ALCI}$ concept $\mathit{OCL\text{-}LiteExpr}^\dagger$ corresponding to these OCL-Lite expressions is respectively

$$C \quad \text{and} \quad \neg C,$$

These OCL-Lite expressions evaluate to true, for a given instance o, if o respectively is and is not of type C.

4. $\mathit{OCL\text{-}LiteExpr} = \mathit{OCL\text{-}LiteExpr}_1$ `and` $\mathit{OCL\text{-}LiteExpr}_2$
The corresponding $\mathcal{ALCI}$ concept $\mathit{OCL\text{-}LiteExpr}^\dagger$ is

$$\mathit{OCL\text{-}LiteExpr}_1^\dagger \sqcap \mathit{OCL\text{-}LiteExpr}_2^\dagger$$

As expected, the $\mathcal{ALCI}$ encoding is the conjunction of the two concepts encoding each sub-expression.

5. $\mathit{OCL\text{-}LiteExpr} = \mathit{OCL\text{-}LiteExpr}_1$ `or` $\mathit{OCL\text{-}LiteExpr}_2$
The corresponding $\mathcal{ALCI}$ concept $\mathit{OCL\text{-}LiteExpr}^\dagger$ is

$$\mathit{OCL\text{-}LiteExpr}_1^\dagger \sqcup \mathit{OCL\text{-}LiteExpr}_2^\dagger$$

The $\mathcal{ALCI}$ encoding is the union of the two concepts encoding each subexpression.

6. $\mathit{OCL\text{-}LiteExpr} = \mathit{OCL\text{-}LiteExpr}_1$ `implies` $\mathit{OCL\text{-}LiteExpr}_2$
The corresponding $\mathcal{ALCI}$ concept $\mathit{OCL\text{-}LiteExpr}^\dagger$ is

$$\neg \mathit{OCL\text{-}LiteExpr}_1^\dagger \sqcup \mathit{OCL\text{-}LiteExpr}_2^\dagger$$

Note that in $\mathcal{ALCI}$ implication is formulated in terms of negation and disjunction.

To further illustrate the mapping, we apply it to some other constraints of our running example. For instance, consider the normalized expression of Constraint 2. This constraint is of kind 1, so its equivalent $\mathcal{ALCI}$ expression is:

$$\text{CriticalEvent} \sqsubseteq \exists\text{Audits}^-.\top$$

A different example of the first kind of OCL-Lite expressions is Constraint 1, which has also been normalized in Section 3.3. This one 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 3. According to the mapping rule, its corresponding $\mathcal{ALCI}$ expression is:

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

Finally, a more complicated example is Constraint 8, which is of kind 5 once it has been normalized. We have to 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 $\mathcal{ALCI}$ expression is:

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

5.3. Correctness of the encoding and complexity results

In this section we first prove that the mapping from OCL-Lite to $\mathcal{ALCI}$ is correct, by showing that there is a direct correspondence between first order interpretations of OCL-Lite constraints and models of the corresponding $\mathcal{ALCI}$ knowledge base. We then show that reasoning on UML/OCL-Lite is an EXPTIME-complete problem.

Theorem 1. Correctness of the OCL-Lite encoding

Let Γ be a set of OCL-Lite constraints and $\mathcal{K}_\Gamma$ its $\mathcal{ALCI}$ encoding. Then, Γ is satisfiable if and only if $\mathcal{K}_\Gamma$ is satisfiable. □

Proof. We have to show that every object that satisfies an OCL-Lite expression is an instance of its corresponding $\mathcal{ALCI}$ concept, and viceversa, i.e., for every *OCL-LiteExpr*, for every interpretation $\mathcal{I}$, and for every object $o \in \Delta^{\mathcal{I}}$, we have that

$$\left(\textit{OCL-LiteExpr}^{f}(x)\right)^{\mathcal{I}}_{[x/o]} \text{ is true} \quad \text{iff} \quad o \in \left(\textit{OCL-LiteExpr}^{\dagger}\right)^{\mathcal{I}},$$

where $[x/o]$ denotes the variable assignment that assigns object o to variable x.

We proceed by induction on the structure of *OCL-LiteExpr*.

1. `[not] oclIsTypeOf(`C`)`
   The FOL formula representing its meaning is $[\neg]C(x)$.
   The corresponding $\mathcal{ALCI}$ concept is: $[\neg]C$.
   Then, $(C(x))^{\mathcal{I}}_{[x/o]}$ is true iff $o \in C^{\mathcal{I}}$. The case of negation easily follows by induction.
2. *OCL-LiteExpr₁* `and` *OCL-LiteExpr₂*
   The FOL formula representing its meaning is $\textit{OCL-LiteExpr}_1^{f}(x) \wedge \textit{OCL-LiteExpr}_2^{f}(x)$.
   The corresponding $\mathcal{ALCI}$ concept is $\textit{OCL-LiteExpr}_1^{\dagger} \sqcap \textit{OCL-LiteExpr}_2^{\dagger}$.
   By induction, the claim easily follows.
   The cases of disjunction and implication are analogous.
3. $p_n ... p_1$`->select(OCL-LiteExpr`$_0$`)->notEmpty()`
   The FOL formula representing its meaning is

$$\exists x_n \cdots x_1 \left(p_n^{f}(x, x_n) \wedge \cdots \wedge p_1^{f}(x_2, x_1) \wedge \textit{OCL-LiteExpr}_0^{f}(x_1)\right).$$

   The corresponding $\mathcal{ALCI}$ concept is $p_n^{\dagger}.(p_{n+1}^{\dagger}.(\cdots(p_1^{\dagger}.(\textit{OCL-LiteExpr}_0^{\dagger}))\cdots))$.
   We prove the claim by induction on the length n of the navigation path.
   Base case ($n = 0$): Then it follows by induction on the structure of OCL-Lite expressions that $\left(\textit{OCL-LiteExpr}_0^{f}(x)\right)^{\mathcal{I}}_{[x/o]}$ is true iff $o \in \left(\textit{OCL-LiteExpr}_0^{\dagger}\right)^{\mathcal{I}}$.
   Inductive case ($n \to n+1$): Let $\textit{OCL-LiteExpr}_n = p_n ... p_{n+1}$`->select(`$\textit{OCL-LiteExpr}_0$`)->notEmpty()`, let p_{n+1} be an additional path item, and let $\textit{OCL-LiteExpr} = p_{n+1}.\textit{OCL-LiteExpr}_n$. We have that $\textit{OCL-LiteExpr}^{f}(x) = \exists x_{n+1}(p_{n+1}^{f}(x, x_{n+1}) \wedge \textit{OCL-LiteExpr}_n^{f}(x_{n+1}))$, and $\textit{OCL-LiteExpr}^{\dagger} = p_{n+1}^{\dagger}.(\textit{OCL-LiteExpr}_n^{\dagger})$. We consider all the different syntactic forms of the path item p_{n+1}.
   - Let p_{n+1} be a role name attached to the *range*(A)-end of some association A. Then, $p_{n+1}^{f}(x, x_{n+1}) = A(x, x_{n+1})$ and $p_{n+1}^{\dagger} = \exists A$. Assume there exists an $o' \in \Delta^{\mathcal{I}}$ such that $A(o', o)$ holds in $\mathcal{I}$ (or, equivalently, $(o, o') \in A^{\mathcal{I}}$).
   - Let p_{n+1} be a role name attached to the *dom*(A)-end of some A. Then, $p_{n+1}^{f}(x, x_{n+1}) = A(x, x_{n+1})$ and $p_{n+1}^{f} = \exists A^{-}$. Assume there exists an $o' \in \Delta^{\mathcal{I}}$ such that $A(o, o')$ holds in $\mathcal{I}$ (or, equivalently, $(o', o) \in A^{\mathcal{I}}$).
   - Let $p_{n+1} =$ `oclAsType(`C`).`r, where r is a role name attached to the *range*(A)-end of some A. Then, $p_{n+1}^{f}(x, x_{n+1}) = C(x) \wedge A(x, x_{n+1})$ and $p_{n+1}^{\dagger} = C \sqcap \exists A$. Assume there exists an $o' \in \Delta^{\mathcal{I}}$ such that $C(o)$ and $A(o', o)$ hold in $\mathcal{I}$ (or, equivalently, $o \in C^{\mathcal{I}}$ and $(o', o) \in A^{\mathcal{I}}$).
   - Let $p_{n+1} =$ `oclAsType(`C`).`r, where r is a role name attached to the *dom*(A)-end of some A. Then, $p_{n+1}^{f}(x, x_{n+1}) = C(x) \wedge A(x_{n+1}, x)$ and $p_{n+1}^{\dagger} = C \sqcap \exists A$. Assume there exists an $o' \in \Delta^{\mathcal{I}}$ such that $C(o)$ and $A(o', o)$ hold in $\mathcal{I}$ (or, equivalently, $o \in C^{\mathcal{I}}$ and $(o', o) \in A^{\mathcal{I}}$).

In all four cases, by inductive hypothesis, $\left(OCL\text{-}LiteExpr_n^f(x_{n+1})\right)^{\mathcal{I}}_{[x_{n+1}/o']}$ is true iff $o' \in \left(OCL\text{-}LiteExpr_n^\dagger\right)^{\mathcal{I}}$. From this the claim follows.
The case where there is no `select` operation, i.e., the OCL-Lite expression has the form $p_n ... p_1$`->notEmpty()`, can be proved analogously.

4. $p_n ... p_1$[`->select(OCL-LiteExpr`$_0$`)`] `->isEmpty()`
The FOL formula representing its meaning is

$$\forall x_n \cdots \forall x_1 \left(\neg p_n^f(x, x_n) \vee \cdots \vee \neg p_1^f(x_2, x_1) \vee \neg OCL\text{-}LiteExpr_0^f(x_1)\right)$$

The corresponding $\mathcal{ALCI}$ concept is $\neg(p_n^\dagger.(\cdots(p_n^\dagger.(OCL\text{-}LiteExpr_0^\dagger))\cdots))$.
The proof is again by induction on the length of the navigation path, and is similar to the previous case. □

Concerning the complexity of reasoning over UML/OCL-Lite schemas we first notice that reasoning just over the UML diagram proposed in this paper is an NP-complete problem. Indeed, the UML language we consider here is a sub-language of the modeling language ER_{bool} which was proved to be NP-complete in [3], and the very same complexity proof applies to the UML language we use. On the other hand, we show that reasoning over UML/OCL-Lite is EXPTIME-complete due to the complexity of the OCL-Lite constraint language.

Theorem 2. Complexity of UML/OCL-Lite

Checking the satisfiability of UML/OCL-Lite conceptual schemas is an EXPTIME-*complete problem.* □

Proof. The upper bound follows from the fact that the $\mathcal{ALCI}$ encoding is correct, and that reasoning in $\mathcal{ALCI}$ is an EXPTIME-complete problem. The lower bound is established by reducing satisfiability of $\mathcal{ALC}$ knowledge bases, which is known to be EXPTIME-complete [4], to satisfiability of OCL-Lite constraints. In particular, we consider so called *primitive* $\mathcal{ALC}$ KBs, i.e., KBs that contain only inclusion assertions of the form

$$C_1 \sqsubseteq C_2, \quad C_1 \sqsubseteq \neg C_2, \quad C_1 \sqsubseteq C_2 \sqcup C_3, \quad C_1 \sqsubseteq \forall A.C_2, \quad C_1 \sqsubseteq \exists A.C_2,$$

where C_1, C_2, C_3 are atomic concepts and A is an atomic role. Satisfiability of primitive $\mathcal{ALC}$ KBs is also EXPTIME-complete, as proved in [7].

To encode inclusion assertions of the form

$$C_1 \sqsubseteq C_2, \quad C_1 \sqsubseteq \neg C_2, \quad C_1 \sqsubseteq C_2 \sqcup C_3,$$

we use the Boolean operators of OCL-Lite. Let us show how to encode the ∀ and ∃ constructs. For each DL role, A, we introduce a binary association, A, with both domain and range untyped (i.e., $dom(A) = range(A) = \top$) and with role name `rng`$_A$ attached to its range. Then:

- $C_1 \sqsubseteq \exists A.C_2$ is encoded as:

  `context` C_1 `inv : rng`$_A$`->select(oclIsTypeOf(`C_2`))->notEmpty()`

- $C_1 \sqsubseteq \forall A.C_2$ is encoded as:

  `context` C_1 `inv : rng`$_A$`->select(not oclIsTypeOf(`C_2`))->isEmpty()`

It is easy to check that the encoding is correct by considering the semantics of OCL-Lite constraints, and considering that every class is a sub-class of $\top$ (thus, C_2 is reachable from C_1 by the role name `rng`$_A$). □

6. Incorporating identification constraints

An *identification constraint* for a class C states the set of properties of C that are unique for every specific instance of, and thus allow one to identify such an instance. Identification constraints are very frequently used in conceptual modeling, but cannot be specified in $\mathcal{ALCI}$. Moreover, the impossibility to specify this kind of constraints in $\mathcal{ALCI}$ implies that association classes and n-ary associations, also very common in conceptual modeling, cannot be encoded in this DL.

Since virtually every UML conceptual schema includes identifiers for its classes, and also association classes or n-ary associations (with or without an association class), we find essential that our approach is able to deal with these constructs. In order to incorporate them, we must make sure that they preserve the finite model property, and we will show that this is the case.

The three constructs we have mentioned (class identifiers, association classes, and n-ary associations) require the ability to define identification constraints. In particular, to uniquely identify each instance of a class C we can add an identification constraint composed by attributes of the class as long as their type belongs to an infinite domain (such as integer, real, string, etc.). Since we know that the rest of the schema fits in $\mathcal{ALCI}$ and this language enjoys the FMP, if there exists an instantiation satisfying the schema but violating the identification constraints, then, it will have a finite number of elements. Thus, since all the domains are infinite, there necessarily exists another finite instance of the schema in which every instance of C has a different value for those attributes that define the identifier. This shows that class identifiers do not destroy the FMP, and moreover that they can safely be ignored when checking satisfiability in UML/OCL-Lite.

Similarly, we show that we can add association classes and n-ary associations (with or without an association class) without losing the finite model property. These constructs cannot be directly encoded in $\mathcal{ALCI}$, but we can express their semantics by transforming the schema into an equisatisfiable one containing only binary associations and specifying additional constraints. Both binary associations with an association class and n-ary associations (with or without an association class) can be treated uniformly by means of reification. More precisely, an association A of arity n, with $n \geq 2$, with or without an association class, can be represented by means of a class A representing the association and n binary functional associations $R_1,\ldots,R_n$ that link A with each of the classes defining the original association, as shown in Fig. 3. This class diagram, together with the constraint stating that the combination of the n binary associations is an identifier for the class A, has the same semantics as an n-ary association between classes $C_1,\ldots,C_n$.

Importantly, neither the identification constraint nor the cardinalities stating the functionality of each R_i, i.e., that every instance of A participates at most once in each of the binary associations R_i, can be expressed in $\mathcal{ALCI}$. However, in this particular case, it is possible to show that allowing for the identification constraint and the functionalities preserves the FMP. Regarding the functionalities, one can adopt a transformation into $\mathcal{ALCI}$ similar to the one proposed in [18,14], which encodes functionality by means of a finite instantiation of an axiom schema. Roughly speaking, the axiom schema enforces that, if in an object o a role R is supposed to be functional, then, for *each possible concept* C, if o satisfies $\exists R.C$, then it satisfies also $\forall R.C$. As shown in [18], it is sufficient to enforce the instances of this axiom schema in which C belongs to a finite set of "relevant" concepts. Intuitively, the relevant concepts are all those that appear as sub-concepts in the DL TBox encoding the UML class diagram and the OCL constraints, and they are polynomially many. In our case, the roles R_i corresponding to the binary associations introduced during reification, are globally functional. Hence, such a transformation is simpler than the one proposed in [14], and requires only to add for each role R_i and for each relevant concept C, an inclusion assertion of the form $\exists R_i.C \sqsubseteq \forall R_i.C$. Such additional inclusion assertions ensure that all individuals connected to an instance of the reified association (class) through a role R_i enjoy the same properties, and hence can be collapsed into a single individual, thus enforcing the satisfaction of functionality. Notice that, differently from the general case considered in [14], due to the fact that the functionality constructor is used only for reification, the transformation that removes it actually preserves the FMP. Regarding the identification constraint, the explanation is analogous to the one adopted in [16] to show the correctness of reification. Intuitively, consider a model $\mathcal{I}$ containing two instances o_1, o_2 of the reified association class A that are connected to exactly the same individuals via roles R_1, …, R_n. By exploiting the disjoint union model property of DLs, one can construct the model that is the union of $\mathcal{I}$ with a copy $\mathcal{I}'$ of $\mathcal{I}$. The copy $\mathcal{I}'$ will contain individuals o'_1 and o'_2 corresponding to o_1 and o_2. By "swapping" the two objects connected to o_2 and o'_2 via one of the roles R_i, say R_n, one gets that the four objects o_1, o'_1, o_2, o'_2 do not constitute anymore a violation of the identification constraint. Hence, by starting from a finite model, containing (a finite number of) violations of an identification constraint, one can apply this construction repeatedly and obtain a new model, again finite, without any violations.

So, we can conclude that we can consider class identifiers, association classes and n-ary associations while preserving the FMP. Thus, reasoning on UML/OCL-Lite schemas containing these constructs can be done in finite time guaranteeing completeness with any kind of approach that admits this expressiveness.

7. Reasoning on UML/OCL-Lite schemas using current reasoners

The aim of this section is to show how current tools may be effectively used to provide reasoning support to check a set of properties on UML/OCL-Lite schemas. We have chosen two tools that follow different approaches: Pellet, a DL reasoner that guarantees completeness and decidability by limiting expressiveness of the constraints, and SVT_E[20], a semidecision procedure which does not limit the expressiveness but does not terminate in some cases due to the undecidability of reasoning with general constraints.

The FMP enjoyed by UML/OCL-Lite schemas guarantees that DL tools, instead of unrestricted satisfiability as in the general case, will check finite satisfiability, i.e., the relevant notion in conceptual modeling. Regarding the semidecision procedure SVT_E, reasoning on UML/OCL-Lite schemas guarantees termination in all cases.

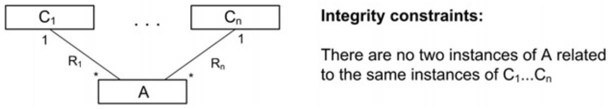

C1
. . .
Cn
1
1
R1
Rn
*
*
A
Integrity constraints:
There are no two instances of A related to the same instances of C1...Cn

Fig. 3. Reification of an n-ary association, with or without an association class.

1. $\top \sqsubseteq \forall \mathsf{Organizes}.\mathsf{Event} \sqcap \forall \mathsf{Organizes}^{-}.\mathsf{Person}$
2. $\top \sqsubseteq \forall \mathsf{Audits}.\mathsf{Event} \sqcap \forall \mathsf{Audits}^{-}.\mathsf{Person}$
3. $\top \sqsubseteq \forall \mathsf{ResponsibleFor}.\mathsf{CriticalEvent} \sqcap \forall \mathsf{ResponsibleFor}^{-}.\mathsf{Person}$
4. $\top \sqsubseteq \forall \mathsf{HeldWith}.\mathsf{Event} \sqcap \forall \mathsf{HeldWith}^{-}.\mathsf{Event}$
5. $\top \sqsubseteq \forall \mathsf{SponsoredBy}.\mathsf{Company} \sqcap \forall \mathsf{SponsoredBy}^{-}.\mathsf{Event}$
6. $\mathsf{Person} \sqsubseteq \exists \mathsf{Organizes}.\top$
7. $\mathsf{CriticalEvent} \sqsubseteq \exists \mathsf{ResponsibleFor}^{-}.\top$
8. $\mathsf{CriticalEvent} \sqsubseteq \mathsf{Event}$
9. $\mathsf{Person} \sqsubseteq \exists \mathsf{Organizes}.(\mathsf{CriticalEvent} \sqcap \neg\exists \mathsf{SponsoredBy}.\top)$
10. $\mathsf{CriticalEvent} \sqsubseteq \exists \mathsf{Audits}^{-}.\top$
11. $\mathsf{Event} \sqsubseteq \neg\exists \mathsf{HeldWith}.\top \sqcup \neg \mathsf{CriticalEvent}$
12. $\mathsf{Person} \sqsubseteq \neg\exists \mathsf{Organizes}.\neg\exists \mathsf{Audits}^{-}.\top$
13. $\mathsf{Person} \sqsubseteq \neg\exists \mathsf{Audits}.\neg\exists \mathsf{SponsoredBy}.\top$
14. $\mathsf{CriticalEvent} \sqsubseteq \exists \mathsf{ResponsibleFor}^{-}.\exists \mathsf{Organizes}.\top$
15. $\mathsf{Event} \sqsubseteq \neg\exists \mathsf{Organizes}^{-}.\top \sqcup \exists \mathsf{SponsoredBy}.\top$
16. $\mathsf{Company} \sqsubseteq \exists \mathsf{SponsoredBy}^{-}.\mathsf{CriticalEvent} \sqcup$
    $\exists \mathsf{SponsoredBy}^{-}.(\neg\exists \mathsf{HeldWith}.\top \sqcup \exists \mathsf{HeldWith}.\neg\exists \mathsf{SponsoredBy}.\top)$

Fig. 4. $\mathcal{ALCI}$ The knowledge base encoding the UML schema in Fig. 1 and the OCL-Lite constraints in Fig. 2.

Our aim in the section is to show how reasoners developed in the Artificial Intelligence community can be used in CASE tools to help the user to check quality criteria of a conceptual model in an automatic way. In particular, we show examples covering the following quality criteria for an UML/OCL-Lite schema:

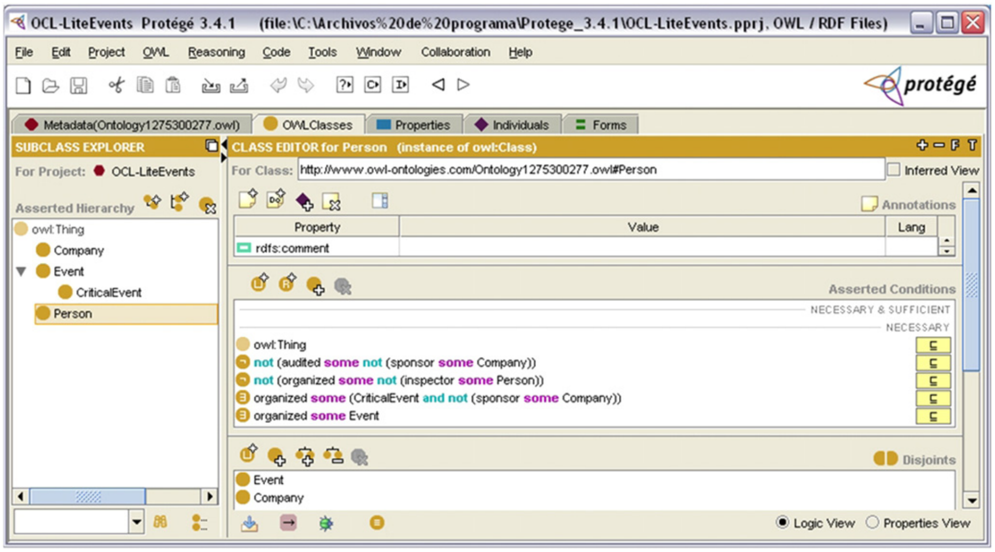
- Class and schema satisfiability;
- OCL-Lite constraint redundancy/entailment;
- explanation of inconsistencies and redundancies/entailments.

Furthermore, we show that current reasoners can be used to reason over UML/OCL-Lite, and that the performance of these tools on our running example is very similar, despite following completely different approaches.

7.1. Using a Description Logics reasoner

We will use the Pellet reasoner included in Protégé version 3.4.1, which implements OWL-DL (a language strictly more expressive than $\mathcal{ALCI}$). The whole knowledge base obtained from the encoding of the UML/OCL-Lite schema of our running example can be captured in Protégé. The $\mathcal{ALCI}$ knowledge base, $\mathcal{K}$, encoding the UML schema in Fig. 1 and the OCL-Lite constraints in Fig. 2 is shown in Fig. 4 (see Sections 5.1 and5.2).

Fig. 5 shows the interface of Protégé, in particular, it shows the *OWLClasses* tab, where the concepts of the ontology are defined. This tab is divided into two main frames: the *Subclass Explorer* at the left shows the hierarchy of concepts, while the *Class Editor* allows one to assert conditions that the concept selected in the *Subclass Explorer* must satisfy. The *Class Editor* is, in turn, divided into three parts, called *Annotations*, *Asserted Conditions*, and *Disjoints*.


OCL-LiteEvents Protégé 3.4.1 (file:\C:\Archivos%20de%20programa\Protege_3.4.1\OCL-LiteEvents.pprj, OWL / RDF Files)
File Edit Project OWL Reasoning Code Tools Window Collaboration Help
protégé
Metadata(Ontology1275300277.owl)
OWLClasses
Properties
Individuals
Forms
SUBCLASS EXPLORER
CLASS EDITOR for Person (instance of owl:Class)
For Project: OCL-LiteEvents
Asserted Hierarchy
owl:Thing
Company
Event
CriticalEvent
Person
Inferred View
Annotations
Property
Value
Lang
rdfs:comment
Asserted Conditions
NECESSARY & SUFFICIENT
NECESSARY
Disjoints
Logic View
Properties View

For Class: http://www.owl-ontologies.com/Ontology1275300277.owl#Person

```
owl:Thing                                                          ⊑
not (audited some not (sponsor some Company))                      ⊑
not (organized some not (inspector some Person))                   ⊑
organized some (CriticalEvent and not (sponsor some Company))      ⊑
organized some Event                                               ⊑
```

Disjoints: Event, Company

(● Logic View ○ Properties View)

Fig. 5. The schema introduced in Protégé.

To introduce the $\mathcal{ALCI}$ formalization of the schema into Protégé, the syntax of the assertions had to be adapted to OWL-DL. In particular, as can be seen in the *Subclass Explorer* in Fig. 5, we have defined three OWL classes that are direct subclasses of `owl:Thing`, namely Company, Event, and Person. We have specified that they are mutually disjoint, as it is usually assumed in UML conceptual schemas. For instance, the fact that Person is disjoint from Company and Event can be seen in the *Disjoints* area at the bottom of the figure. We have also specified the class CriticalEvent as a subclass of Event.

Each association has been represented by two properties, specifying their domain and range and the inverse relationships between them. For instance, audited is a property with domain Person and range Event, and inspector is the inverse of audited. These two properties correspond to the roles Audits and Audits^{-} in the $\mathcal{ALCI}$ formalization of our example. They can be introduced in Protégé in the *Properties* tab, not shown in Fig. 5.

Now, both the cardinalities and the OCL constraints have to be introduced as *Asserted Conditions* of the corresponding concept. For instance, as can be seen in Fig. 5, several necessary conditions have been specified for the class Person, corresponding to the $\mathcal{ALCI}$ assertions in which Person is subsumed by a concept encoding some condition. The first of them corresponds to the OCL Constraint 5, the second is Constraint 4, the third encodes Constraint 1, and the last one is the cardinality `1..*` of *organized* (assertion 6 in the $\mathcal{ALCI}$ formalization of the schema given in Fig. 4).

Once we have introduced the UML class diagram and the OCL-Lite constraints into Protégé, we can perform the reasoning tasks we mentioned. The reasoning task in which we are interested in is *consistency*, since checking for either *redundancy* or *logical implication* can be reduced to a consistency check. First, we check the consistency of all the concepts we have defined. As can be seen in Fig. 6, Person and CriticalEvent are inconsistent. As we have shown in the introduction, the reason for these inconsistencies is that Constraints 1, 2, and 5 are in contradiction. This additional information is what we called *explanation*. The current version of Protégé does not provide such a reasoning service. This means that the schema is incorrect, and should be fixed. We do it by replacing Constraint 1 with the following one:

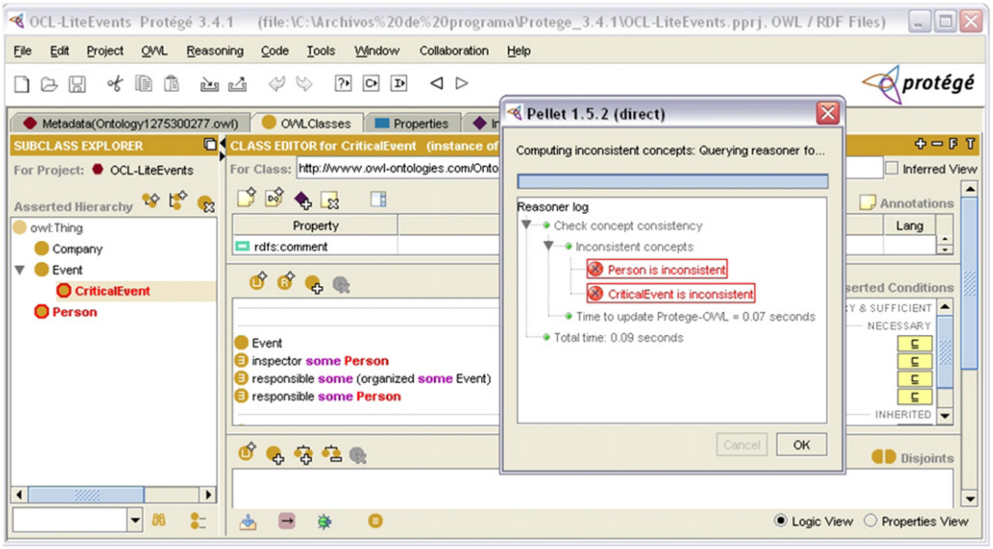
```
context Person inv : organized −>exists(oclIsTypeOf(CriticalEvent))
```

That is, each person must organize at least one critical event. This new constraint makes all classes consistent.

Now that the schema is consistent, we can check other properties. For instance, we are interested in checking the *redundancy* of the constraints in the schema. A constraint is redundant if it is *entailed* by the other constraints and, thus, it can be removed. We can check redundancy in Protégé by checking the inconsistency of a concept that corresponds to the negation of the constraint. That is, let $C \sqsubseteq D$ be the $\mathcal{ALCI}$ encoding of one of the OCL-Lite constraints in the schema and let $\mathcal{K}$ be the knowledge base encoding the whole UML/OCL-Lite model. Then, the constraint is redundant iff $\mathcal{K} \setminus \{C \sqsubseteq D\} \models C \sqsubseteq D$ which, in turn, holds iff the concept $C \sqcap \neg D$ is unsatisfiable w.r.t. $\mathcal{K} \setminus \{C \sqsubseteq D\}$. The same technique can be applied to check whether the UML/OCL-Lite model verifies a new constraint. In this case, we check whether $\mathcal{K}$ entails the new constraint, by transforming this check into a concept satisfiability check.

For instance, Constraint 7 states that those events that have an organizer also have a sponsor. To check the redundancy of this constraint, we can introduce a new concept, notIC7, as a subclass of the concept Event, and add an inclusion assertion corresponding to the negation of the original assertion encoding Constraint 7, which, in turn, must be removed from $\mathcal{K}$, i.e.:

$$\mathsf{notIC7} \sqsubseteq \mathsf{Event}$$
$$\mathsf{notIC7} \sqsubseteq \exists \mathsf{Organizes}^{-}.\top \sqcap \neg \exists \mathsf{SponsoredBy}.\top$$


OCL-LiteEvents Protégé 3.4.1 (file:\C:\Archivos%20de%20programa\Protege_3.4.1\OCL-LiteEvents.pprj, OWL / RDF Files)
File Edit Project OWL Reasoning Code Tools Window Collaboration Help
protégé
Metadata(Ontology1275300277.owl)
OWLClasses
Properties
SUBCLASS EXPLORER
For Project: OCL-LiteEvents
Asserted Hierarchy
owl:Thing
Company
Event
CriticalEvent
Person
CLASS EDITOR for CriticalEvent (instance of
For Class: http://www.owl-ontologies.com/Onto
Property
rdfs:comment
Lang
Annotations
Inferred View
serted Conditions
Y & SUFFICIENT
NECESSARY
INHERITED
Disjoints
Logic View
Properties View
Pellet 1.5.2 (direct)
Cancel
OK

Event
inspector some Person
responsible some (organized some Event)
responsible some Person

```
Computing inconsistent concepts: Querying reasoner fo...

Reasoner log
▼ Check concept consistency
    ▼ Inconsistent concepts
        Person is inconsistent
        CriticalEvent is inconsistent
      Time to update Protege-OWL = 0.07 seconds
  Total time: 0.09 seconds
```

Fig. 6. Results of checking consistency using the Pellet reasoner in Protégé.

As can be checked, the concept notIC7 is inconsistent w.r.t. $\mathcal{K}\backslash\{\mathsf{Event}\sqsubseteq\neg\exists\mathsf{Organizes}^{-}.\top\sqcup\exists\mathsf{SponsoredBy}.\top\}$, which means that Constraint 7 is redundant. Thus, the OCL-Lite Constraint 7 can be removed preserving the semantics of the schema. We can run the same redundancy check for Constraint 6, and the result will be that Constraint 6 is also redundant, so it can be eliminated from the schema. If we check redundancy of the remaining constraints, we see that they are not redundant (i.e., their negation is consistent).

Finally, we will check some additional properties on the schema to see whether it represents correctly the intended domain. To do this, we define a new concept satisfying a certain property and check whether it is consistent. If this is the case, then the property may hold in the schema. For instance, we may be interested in checking whether a critical event may be organized by some person that is not responsible for any critical event. Thus, we define a concept Property1 as can be seen in Fig. 7(a). As shown, this property is consistent, which means that the situation that it formalizes is accepted by the schema. Another interesting question could be whether it is possible that a person organizes an event that does not have an inspector (see Fig. 7(b)). This time, this property is not consistent, which in fact means that all the events have an inspector, despite the cardinality `*` of the association end `inspector` specified in the UML class diagram. This answer indicates that either the cardinality should be changed to `1..*` or that the rest of the constraints should be weakened so that they correspond to the specified cardinality.

7.2. Using SVT$_E$

We have shown in [35] that we may use the CQC-Method [21] to reason on a UML/OCL conceptual schema. This method performs query containment tests on deductive database schemas, and we can use it to reason on conceptual schemas by translating the class diagram and the constraints into the logic representation it handles. The most recent implementation of this method is the SVT$_E$ tool [20], which is the one we will use to reason on our running example. SVT$_E$ allows for checking whether a given schema satisfies a set of desirable properties (i.e., it checks the satisfiability of a schema), or whether a class in the schema is consistent (i.e., class satisfiability check). Each property is specified in terms of a certain goal to attain defined as a conjunction of literals. When a property is satisfied, SVT$_E$ provides a sample instantiation of the schema satisfying the property. Otherwise, it gives an *explanation*, i.e., the set of constraints that do not allow the property to be satisfied.

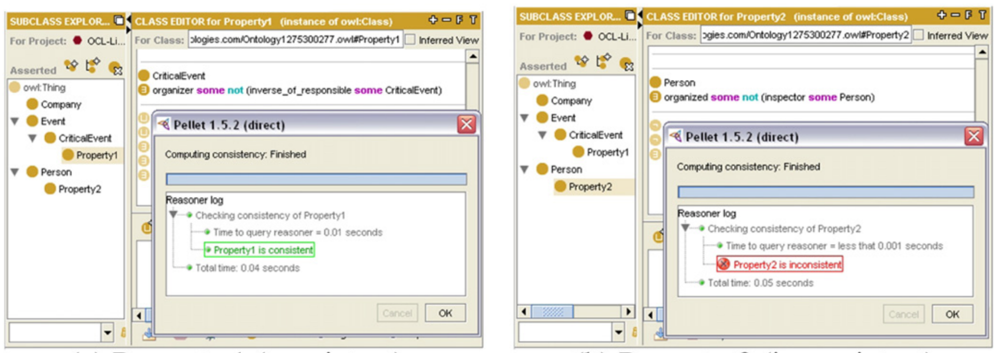
First of all, we must translate the UML/OCL-Lite schema into a logic representation as defined in [35]. The translation we get for our running example can be seen in the left hand side of Fig. 8, which shows a screenshot of the SVT$_E$ tool.

We may be interested to know whether the class `Company` admits at least one instance, i.e., if it is consistent. The goal to attain in this case is $company(C)$ and this is the question we pose to SVT$_E$, as shown in Fig. 8. The answer we get from the tool is that `Company` is consistent since the schema admits an instantiation, $I=\{company(0), event(1), sponsoredBy(1, 0)\}$, where `Company` has a non-empty extension. Note that the two latter instances are required to make the sample instantiation satisfy Constraint 8 and that they are automatically obtained by SVT$_E$. This instantiation is partially shown in Fig. 8 which shows how SVT$_E$ represents the instances of `SponsoredBy`. Clicking on *event* or on *company*, we get the other two instances of I.

In a similar way we check in SVT$_E$ the consistency of class `Person` by asking the question $person(P)$. In this case, we get that the goal is not satisfiable (as expected) and also the *explanation* for this failure shown in Fig. 8. The intuitive meaning of this explanation is the following. According to Constraint 1, each `Person` must organize a non-sponsored `CriticalEvent`. However, Constraint 7 forces all events (critical or not) with an organizer to have a `Sponsor`. Therefore, we get into a contradiction, which entails that `Person` is not consistent since it can never be populated. The third constraint in the explanation is required to formally ensure that the second argument of the association `Organizes` is an `Event`.

In general, there may be alternative explanations that justify the failure of a goal. Note, for instance, that the one obtained by SVT$_E$ is not the same that we gave in the introduction. Alternative explanations, including this latter one, are obtained with SVT$_E$ by clicking on the button '*Compute all minimal explanations*'.

The previous result implies that the schema is incorrect, and it should be necessarily modified. Again, we fix this error by removing from Constraint 1 the requirement that the critical event each person must organize cannot be sponsored.



```
CriticalEvent
organizer some not (inverse_of_responsible some CriticalEvent)

Computing consistency: Finished

Reasoner log
Checking consistency of Property1
    Time to query reasoner = 0.01 seconds
    Property1 is consistent
Total time: 0.04 seconds
```

(a) Property 1 (consistent)

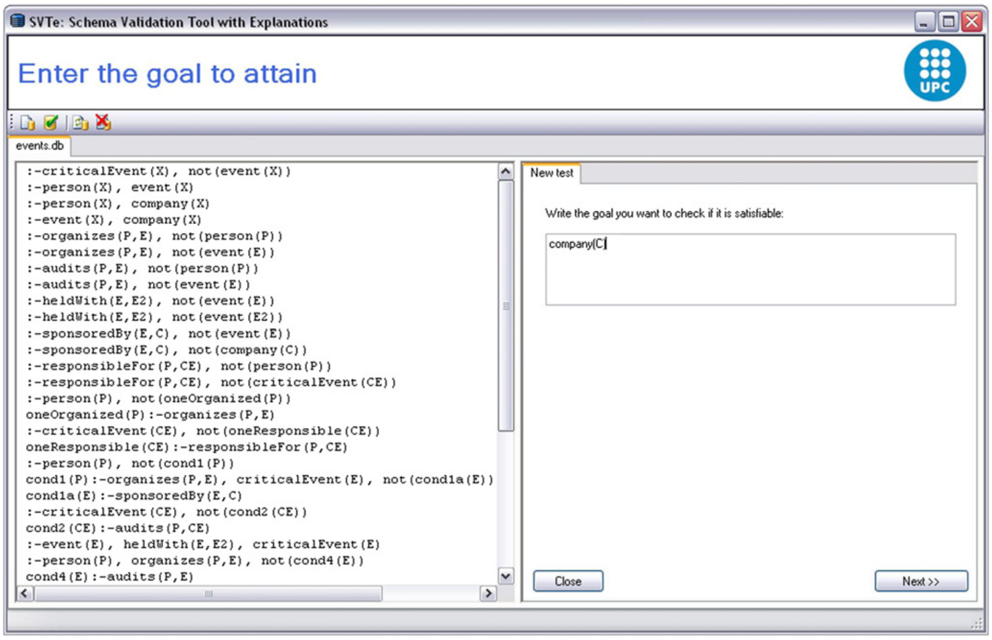
```
Person
organized some not (inspector some Person)

Computing consistency: Finished

Reasoner log
Checking consistency of Property2
    Time to query reasoner = less that 0.001 seconds
    Property2 is inconsistent
Total time: 0.05 seconds
```

(b) Property 2 (inconsistent)

Fig. 7. Results of checking user-defined properties in Protégé.



```
:-criticalEvent(X), not(event(X))
:-person(X), event(X)
:-person(X), company(X)
:-event(X), company(X)
:-organizes(P,E), not(person(P))
:-organizes(P,E), not(event(E))
:-audits(P,E), not(person(P))
:-audits(P,E), not(event(E))
:-heldWith(E,E2), not(event(E))
:-heldWith(E,E2), not(event(E2))
:-sponsoredBy(E,C), not(event(E))
:-sponsoredBy(E,C), not(company(C))
:-responsibleFor(P,CE), not(person(P))
:-responsibleFor(P,CE), not(criticalEvent(CE))
:-person(P), not(oneOrganized(P))
oneOrganized(P):-organizes(P,E)
:-criticalEvent(CE), not(oneResponsible(CE))
oneResponsible(CE):-responsibleFor(P,CE)
:-person(P), not(cond1(P))
cond1(P):-organizes(P,E), criticalEvent(E), not(cond1a(E))
cond1a(E):-sponsoredBy(E,C)
:-criticalEvent(CE), not(cond2(CE))
cond2(CE):-audits(P,CE)
:-event(E), heldWith(E,E2), criticalEvent(E)
:-person(P), organizes(P,E), not(cond4(E))
cond4(E):-audits(P,E)
```

Write the goal you want to check if it is satisfiable:

company(C)

Fig. 8. Checking a property with SVT$_E$.

SVT$_E$ also allows one to identify *redundant* constraints. Redundancy of a constraint may be checked in SVT$_E$ by removing it from the schema and trying to attain the negation of its goal. For instance, the goal that checks whether Constraint 3 is redundant is: *event*(E), *heldWith*(E, $E2$), *criticalEvent*(E). As before, we get that Constraints 6 and 7 are redundant, while the above Constraint 3 is not. Moreover, SVT$_E$ provides us with an explanation each time a redundancy is found. In particular, Constraint 6 is redundant because of the cardinality constraints `1..*` of the roles `responsible` and `organized`. They state that each critical event must have at least a responsible that, in turn, must organize some event. So, these cardinality constraints alone ensure the same condition as the one stated by Constraint 6. Constraint 7 is also redundant since it is impossible that an event that has an organizer is not sponsored. The reason is that Constraint 4 ensures that all organized events are audited, and 5 guarantees that audited events have a sponsor. As a consequence, organized events necessarily have a sponsor, which is what Constraint 7 states. Summarizing, we can remove the redundant Constraints 6 and 7 from the schema so that it becomes simpler without changing its semantics.

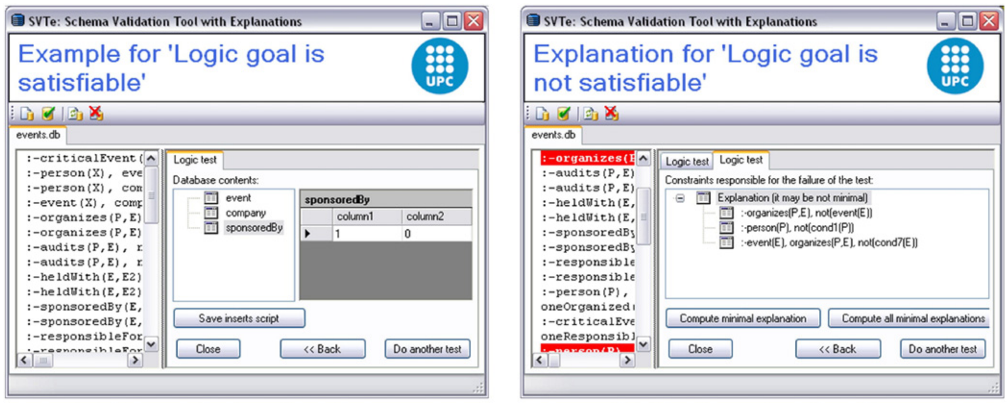
SVT$_E$ allows also for asking ad-hoc questions to check whether the schema specifies what the designer intended, i.e., if it is compliant with the requirements. For instance, to check whether a critical event may be organized by some person that is not responsible for any critical event, we should consider the goal: *criticalEvent*(CE), *organizes*(P, CE), *not*(*resp*(P)), where *resp*(P) should be defined by means of the rule *responsibleFor*(P, CE), *criticalEvent*(CE). SVT$_E$ returns a positive answer in this case, proving that the schema satisfies the requirement, and a set consisting of 10 instances that fulfill this situation. Finally, the question whether a person may organize an event that does not have an inspector can also be answered by SVT$_E$. Now, the answer we get is negative since the property we are checking is in contradiction with Constraint 4 (the explanation provided by SVT$_E$). Therefore, either this requirement is wrong or we must modify again the schema to make it fulfilled (Fig. 9).

7.3. On the performance of current reasoners

We have also analyzed how the two chosen reasoners perform on our example. Although the conceptual schema considered is quite simple, the results obtained are promising, since each reasoning task takes much less than one second although the schema contains 8 constraints to encode the semantics of the UML class diagram, and 8 additional arbitrary OCL constraints.

We have checked the consistency of each class, the (non) redundancy/entailment of each constraint and the user-defined properties explained in the previous subsections. Some of these properties are satisfied by the schema while others are not. All of them have been successfully checked by both reasoners. The time spent[4] in each reasoning task goes from a minimum of 0,01/0,02 s (consistency of classes `Event` and `Company`, entailment of Constraint 8) to a maximum of 0,04/0,05 s (consistency

[4] We have executed the experiments on an Intel Pentium 4, 3.2 GHz machine with 1 GB RAM and Windows XP (SP3).


SVTe: Schema Validation Tool with Explanations

Example for 'Logic goal is satisfiable'

events.db

```
:-criticalEvent(
:-person(X), eve
:-person(X), com
:-event(X), comp
:-organizes(P,E)
:-organizes(P,E)
:-audits(P,E), r
:-audits(P,E), r
:-heldWith(E,E2)
:-heldWith(E,E2)
:-sponsoredBy(E,
:-sponsoredBy(E,
:-responsibleFor
```

Logic test

Database contents:

- event
- company
- sponsoredBy

sponsoredBy

	column1	column2
▶	1	0

SVTe: Schema Validation Tool with Explanations

Explanation for 'Logic goal is not satisfiable'

events.db

```
:-organizes(E
:-audits(P,E)
:-audits(P,E)
:-heldWith(E,
:-heldWith(E,
:-sponsoredBy
:-sponsoredBy
:-responsible
:-responsible
:-person(P),
oneOrganized
:-criticalEve
oneResponsibl
```

Logic test | Logic test

Constraints responsible for the failure of the test:

- Explanation (it may be not minimal)
  - :-organizes(P,E), not(event(E))
  - :-person(P), not(cond1(P))
  - :-event(E), organizes(P,E), not(cond7(E))

(a) Instantiation for *Company* (consistent)

(b) Explanation for *Person* (inconsistent)

Fig. 9. Results of checking consistency in SVT$_E$.

of class `CriticalEvent`, entailment of Constraints 4 and 5, user-defined properties). These are just preliminary tests showing that the current technology can be used in practice to check properties of UML/OCL-Lite schemas.

8. Conclusions and further work

We have identified fragments of both UML and OCL that guarantee finite reasoning while being at the same time significantly expressive. The UML fragment maintains almost all the modeling constructs, but the at-most cardinality restriction on both relationships and attributes is disallowed. This restriction is crucial to preserve the FMP property when reasoning over UML schemas. Concerning OCL, since full OCL is undecidable we devised here a new fragment to gain both decidability and FMP. We called such a new fragment OCL-Lite. We proved that the proposed UML/OCL-Lite fragment enjoys the FMP by showing a satisfiability preserving mapping from UML/OCL-Lite to the DL $\mathcal{ALCI}$. The mapping guarantees that every satisfiable set of OCL-Lite constraints admits a finite model. To our knowledge, this is the first attempt to encode OCL constraints in DLs.

One of the most relevant side effects of the work presented in this paper is the possibility to use a DL reasoner to reason on UML/OCL-Lite schemas. We have shown how current tools may be effectively used to provide reasoning support in order to automatically check a set of properties of UML/OCL-Lite schemas, in particular, class and schema satisfiability, OCL-Lite constraint redundancy/entailment, and explanation of inconsistencies and redundancies/entailments. We have chosen two tools that follow different paradigms: Pellet, a DL reasoner, and SVT$_E$, a semidecision procedure. Due to the FMP enjoyed by UML/OCL-Lite schemas, it is guaranteed that Pellet is able to check finite satisfiability, while SVT$_E$ will always terminate. Such reasoning capabilities can be incorporated into existing CASE tools to extend their functionalities.

As further work we plan to devise maximal decidable fragments of OCL that can be encoded using expressive (but decidable) DLs. In particular, we are considering more powerful DLs starting from $\mathcal{ALCQI}$ to the more complex $\mathcal{SHROIQ}$ [26]. These new fragments will lose the FMP property but they will retain decidability, and they will also allow to reintroduce full cardinality restrictions in UML schemas. An interesting open issue concerned with the FMP is to investigate the problem of finite model reasoning when encoding into the powerful $\mathcal{SHROIQ}$ language. Furthermore, although it is known that finite model reasoning is ExpTime-complete for $\mathcal{ALCQI}$[30], there are currently no DL reasoners able to handle such a form of reasoning. Another direction is more applicative and concerned with an experimental evaluation to compare the efficiency and correctness of different techniques currently developed to check schema properties of both UML and OCL constraints.

Acknowledgments

This work has been partly supported by the Ministerio de Ciencia e Innovación under the projects TIN2008-03863 and TIN2008-00444, Grupo Consolidado, and the FEDER funds; and by the project OntoRule, FP7 Integrating Project (IP), no. 231875, co-funded by the European Union.

References

[1] K. Anastasakis, B. Bordbar, G. Georg, I. Ray, On challenges of model transformation from UML to alloy, Software and Systems Modeling 9 (1) (2010) 69–86.

[2] A. Artale, D. Calvanese, A. Ibáñez-García, Full satisfiability of UML class diagrams, Proc. of the 29th Int. Conf. on Conceptual Modeling (ER 2010), Volume 6412 of LNCS, Springer, 2010, pp. 317–331.

[3] A. Artale, D. Calvanese, R. Kontchakov, V. Ryzhikov, M. Zakharyaschev, Reasoning over extended ER models, Proc. of the 26th Int. Conf. on Conceptual Modeling (ER 2007), Volume 4801 of LNCS, Springer, 2007, pp. 277–292.
[4] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003.
[5] B. Beckert, U. Keller, P.H. Schmitt, Translating the Object Constraint Language into first-order predicate logic, Proc. of the VERIFY Workshop at Federated Logic Conferences (FLoC), 2002, pp. 113–123.
[6] D. Beimel, M. Peleg, Using OWL and SWRL to represent and reason with situation-based access control policies, Data and Knowledge Engineering (DKE) 70 (6) (2011) 596–615.
[7] D. Berardi, D. Calvanese, G. De Giacomo, Reasoning on UML class diagrams, Artificial Intelligence 168 (1–2) (2005) 70–118.
[8] A.D. Brucker, B. Wolff (Eds.), The HOL-OCL Book, Swiss Federal Institute of Technology, 2006.
[9] F. Bry, R. Manthey, Checking consistency of database constraints: a logical basis, Proc. of the Twelth Int. Conf. on Very Large Data Bases (VLDB'86), 1986, pp. 13–20.
[10] J. Cabot, R. Clariso, D. Riera, Verification of UML/OCL class diagrams using constraint programming, Proc. of the Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa 2008), 2008.
[11] M. Cadoli, D. Calvanese, G. De Giacomo, T. Mancini, Finite model reasoning on UML class diagrams via constraint programming, Proc. of the 10th Congress of the Italian Assoc. for Artificial Intelligence (AI*IA 2007), Volume 4733 of LNAI, Springer, 2007, pp. 36–47.
[12] D. Calvanese, Finite model reasoning in description logics, Proc. of the 5th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'96), 1996, pp. 292–303.
[13] D. Calvanese, G. De Giacomo, Expressive description logics, in: F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003, pp. 178–218.
[14] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, Reasoning in expressive description logics, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, Volume II, Chapter 23, Elsevier Science Publishers, 2001, pp. 1581–1634.
[15] D. Calvanese, M. Lenzerini, On the interaction between ISA and cardinality constraints, Proc. of the 10th IEEE Int. Conf. on Data Engineering (ICDE'94), 1994, pp. 204–213.
[16] D. Calvanese, M. Lenzerini, D. Nardi, Unifying class-based representation formalisms, Journal of Artificial Intelligence Research (JAIR) 11 (1999) 199–240.
[17] S.S. Cosmadakis, P.C. Kanellakis, M.Y. Vardi, Polynomial-time implication problems for unary inclusion dependencies, Journal of the ACM 37 (1) (1990) 15–46.
[18] G. De Giacomo. Decidability of Class-Based Knowledge Representation Formalisms. PhD Thesis, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", 1995.
[19] S. Dupuy, Y. Ledru, M. Chabre-Peccoud, An overview of RoZ: a tool for integrating UML and Z specifications, Proc. of the 12th Int. Conf. on Advanced Information Systems Engineering (CAiSE 2000), Volume 1789 of LNCS, Springer, 2000, pp. 417–430.
[20] C. Farré, G. Rull, E. Teniente, T. Urpí, SVTe: a tool to validate database schemas giving explanations, Proc. of the 1st Int. Workshop on Testing Database Systems (DBTest 2008), ACM Press, 2008.
[21] C. Farre, E. Teniente, T. Urpí, Checking query containment with the CQC method, Data and Knowledge Engineering (DKE) 53 (2) (2005) 163–223.
[22] P.R. Fillottrani, E. Franconi, S. Tessaris, The new ICOM ontology editor, Proc. of the 19th Int. Workshop on Description Logic (DL 2006), Volume 189 of CEUR Electronic Workshop Proceedings, 2006 http://ceur-ws.org/.
[23] A. Formica, Finite satisfiability of integrity constraints in object-oriented database schemas, IEEE Transactions on Knowledge and Data Engineering 14 (1) (2002) 123–139.
[24] M. Gogolla, F. Büttner, M. Richters, USE: a UML-based specification environment for validating UML and OCL, Science of Computer Programming 69 (1–3) (2007) 27–34.
[25] S. Hartmann, Coping with inconsistent constraint specifications, Proc. of the 20th Int. Conf. on Conceptual Modeling (ER 2001), Volume 2224 of LNCS, Springer, 2001, pp. 241–255.
[26] I. Horrocks, O. Kutz, U. Sattler, The even more irresistible $\mathcal{SROIQ}$, Proc. of the 10th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2006), 2006, pp. 57–67.
[27] D. Jackson, Alloy: a lightweight object modelling notation, ACM Transactions on Software Engineering and Methodology 11 (2) (2002) 256–290.
[28] M. Kuhlmann, L. Hamann, M. Gogolla, Extensive validation of OCL models by integrating SAT solvers into USE, Proc. of the 49th Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS 2011), Volume 6705 of LNCS, Springer, 2011, pp. 290–306.
[29] M. Lenzerini, P. Nobili, On the satisfiability of dependency constraints in entity-relationship schemata, Information Systems 15 (4) (1990) 453–461.
[30] C. Lutz, U. Sattler, L. Tendera, The complexity of finite model reasoning in description logics, Information and Computation 199 (2005) 132–171.
[31] MIT Software Design Group, The Alloy Analyzer, http://alloy.mit.edu.
[32] A. Olivé, Conceptual Modeling of Information Systems, Springer, 2007.
[33] OMG, UML 2.2 Superstructure Specification Available at, http://www.uml.org/2009.
[34] OMG, Object Constraint Language. Version 2.2 Available at, http://www.omg.org/spec/OCL/Feb. 2010.
[35] A. Queralt, E. Teniente, Reasoning on UML class diagrams with OCL constraints, Proc. of the 25th Int. Conf. on Conceptual Modeling (ER 2006), Volume 4215 of LNCS, Springer, 2006, pp. 497–512.
[36] A. Queralt, E. Teniente, Decidable reasoning in UML schemas with constraints, Proc. of the 20th Int. Conf. on Advanced Information Systems Engineering (CAiSE 2008), Volume 5074 of LNCS, Springer, 2008, pp. 281–295.
[37] A. Queralt, E. Teniente, Verification and validation of UML conceptual schemas with OCL constraints, ACM Transactions on Software Engineering and Methodology 21 (2) (2011) To appear.
[38] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison Wesley Publ. Co., 1998
[39] C.F. Snook, M.J. Butler, UML-B: formal modeling and design aided by UML, ACM Transactions on Software Engineering and Methodology 15 (1) (2006) 92–122.
[40] M. Wahler, D. Basin, A.D. Brucker, J. Koehler, Efficient analysis of pattern-based constraint specifications, Software and Systems Modeling 9 (2) (2010) 225–255.
[41] J. Warmer, A. Kleppe, The Object Constraint Language. Second Edition. Getting Your Models Ready For MDA, Addison-Wesley, 2003.

Anna Queralt is an assistant professor at the Department of Service and Information System Engineering at the Universitat Politèclnica de Catalunya – BarcelonaTech, where she teaches software engineering. She got her PhD in Computer Science from the same university in 2009. She has been a visiting researcher at the Free University of Bozen-Bolzano. Her research is mainly focused on conceptual modeling and automated reasoning on conceptual schemas.

Dr. Alessandro Artale is an Assistant Professor in the Faculty of Computer Science at the Free University of Bolzano where he teaches graduate and undergraduate courses. He got a PhD in Computer Science from the University of Florence in 1994. He published more than 70 papers in international journals and conferences and as book chapters. He acted as both Chair and PC member both in International Conferences, and as editor of both proceedings and journal's special issues. His research has been funded by the European Community and by National funds. His main research subject concerns description logics, temporal logic, automated reasoning, ontologies and conceptual modeling. A particular emphasis is devoted to the formalization of conceptual modeling tasks in domains with high semantic complexity and characterized by a dynamic aspect.

Diego Calvanese is an associate professor at the KRDB Research Centre for Knowledge and Data, Free University of Bozen-Bolzano, where he teaches graduate and undergraduate courses on theory of computing, formal languages, knowledge bases and databases, and ontologies. His research interests include formalisms for knowledge representation and reasoning, ontology languages, description logics, Semantic Web, conceptual data modeling, data integration, semistructured data management, and service modeling and synthesis. He is actively involved in several national and international research projects in the above areas, and he is the author of more than 200 refereed publications, including ones in the most prestigious international journals and conferences in Databases and Artificial Intelligence. He is one of the editors of the Description Logic Handbook. He is regularly invited to serve on the Program Committees of international conferences in the above mentioned areas and is a member of the editorial board of JAIR.

Ernest Teniente is a full professor at the Department of Service and Information System Engineering at the Universitat Politècnica de Catalunya – BarcelonaTech, where he teaches graduate and undergraduate courses on software engineering and databases. He got his PhD in Computer Science from the same university. He was a visiting researcher at the Politecnico di Milano and at the Universita' di Roma Tre, in Italy. His current research interests are focused on conceptual modeling, automated reasoning on conceptual schemas and data integration. He is author of more than 50 publications in international conferences and journals in the areas of databases and software engineering, and he is regularly invited to serve on the Program Committees of international conferences in these areas.