Reasoning on UML Class Diagrams with OCL Constraints

Anna Queralt and Ernest Teniente

Universitat Politècnica de Catalunya
Dept. de Llenguatges i Sistemes Informàtics
c/ Jordi Girona 1-3, 08034 Barcelona (Catalonia, Spain)
{aqueralt, teniente}@lsi.upc.edu

Abstract. We propose a new approach to check whether a given UML class diagram with its OCL integrity constraints satisfies a set of desirable properties such as schema satisfiability, class liveliness, redundancy of integrity constraints or reachability of partially specified states. Our approach is based on translating both the class diagram and the OCL constraints into a logic representation. Then, we use the CQC Method to verify whether these properties hold for the given diagram and constraints.

1. Introduction

The quality of an information system is largely determined early in the development cycle, i.e. during requirements specification and conceptual modeling. Moreover, errors introduced at these stages are usually much more expensive to correct than those of design or implementation. Thus, it is desirable to prevent, detect and correct errors as early as possible in the development process.

Quality of a conceptual schema (CS) can be seen from two different points of view. From an external point of view, quality refers to the correctness of the schema regarding the user requirements. This can be validated, for instance, through checking whether the schema specifies the relevant knowledge of the domain. From an internal point of view, quality can be determined by reasoning on the definition of the CS, without taking requirements into account. In this sense, there are some typical reasoning tasks that can be performed on a CS like satisfiability checking, class consistency, etc.

As a simple example, consider the CS in Figure 1 which contains information about employees (distinguishing among rich and average employees) and their categories. The schema contains also four integrity constraints stating conditions that each state of the information base should satisfy.

At first glance, it may seem that the schema is perfectly right. It allows instances of *Employee* (like Employee(John,7000), *Category* (like Category(Sales,6000) and *WorksIn* (like WorksIn(John,Sales)). However, a deeper analysis allows determining that *AverageEmp* will never have any instance since constraint 5 requires the salary of average employees to be lower than 5000 while constraints 2 and 3 assert that it must be higher than 5000. That means that *AverageEmp* is clearly ill specified. Moreover, constraint 4 is redundant (and, thus, it does not need to be defined) since constraints 2 and 3 already guarantee that the salary of a rich employee is higher than 5000.

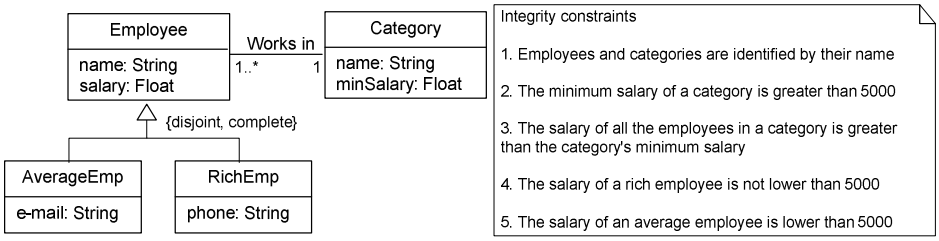

Fig. 1. Conceptual schema about employees and their categories

The previous example illustrates the need to be able to reason on UML CSs to improve information systems quality. In fact, this has been identified as one of the key problems to be solved for achieving the goal of automating information systems building [13].

Several efforts have already been devoted to this problem. There are automatic procedures for the verification of some properties of CSs in Description Logics [2], to check whether a given CS accepts particular system states at a particular time point [9] or to reason about cardinality constraints of the CS [10-12]. However, there are still several open problems in reasoning on conceptual schemas. Probably, the most important one is the lack of methods able to reason about general-purpose integrity constraints like the ones in Figure 1. This is indeed the main goal of this paper.

Hence, the main contribution of our work is to propose a new approach to reason on the structural part of UML CSs, defined by means of the corresponding UML class diagram with its OCL integrity constraints.

Two different kinds of reasoning are provided by our method. On the one hand, it may automatically verify whether the CS satisfies a set of desirable properties such as schema satisfiability, class liveliness (in the example above it would determine that *AverageEmp* is not lively) or redundancy of integrity constraints (like constraint 4 in Figure 1).

On the other hand, it provides the designer with the ability of asking questions to check whether certain conceived goals may be satisfied according to the CS. For instance, if he wants to know whether the CS in Figure 1 accepts a category Marketing with a salary of 8000, our method would answer positively provided that there is at least one employee working on it (required to satisfy the multiplicity constraint of the association *WorksIn*), with a salary higher than 8000 (entailed by constraint 3) and who is a *RichEmp* (because of the complete constraint of the generalization).

This paper is organized as follows. Section 2 presents an overview of our method. Section 3 defines how to translate an UML CS into a logical representation. Section 4 describes how to use the CQC Method to reason on UML CSs. Section 5 reviews related work. Finally, section 6 presents our conclusions and points out future work.

2. Overview of the Method

The structural part of a CS consists of a taxonomy of classes together with their attributes, a taxonomy of associations among entity types, and a set of integrity

constraints over the state of the domain, which define conditions that each state of the information base must satisfy. Those constraints may have a graphical representation or can be defined by means of a particular general-purpose language.

In UML, a structural schema is represented by means of a class diagram, with its graphical constraints, together with a set of user-defined constraints, which can be specified in any language. As proposed in [15], we will assume these constraints are specified in OCL.

The subset of the OCL language we consider consists of those OCL expressions that are used in integrity constraints, not those operations that can only be used in pre or postconditions, such as @pre and oclIsNew. Moreover, we only deal with OCL operations that result in a boolean value. An exception are *select* and *size* that, despite returning a collection and an integer, can also be handled by our method.

In Figure 2 we show the structural schema we will use throughout the paper. It consists of a UML class diagram with three classes, three associations and seven OCL constraints. It states that several employees work in a category, which is managed by one employee. Some employees have a superior, and some employees are bosses.

The OCL constraints provide the class diagram with additional semantics. There are two key constraints (*UniqueEmp* and *UniqueCat*), one for each class. The constraint *ManagerIsWorker* states that the manager of a category must be one of its workers. Constraint *ManagerHasNoSuperior* guarantees that the manager of a category does not work for any other employee. The constraint *BossIsManager* guarantees that a boss is the manager of some category. The next constraint, *BossHasNoSuperior,* states that a boss does not work for any other employee. Finally, constraint *SuperiorOfAllWorkers* states that the workers of a category managed by a boss must work for that boss.

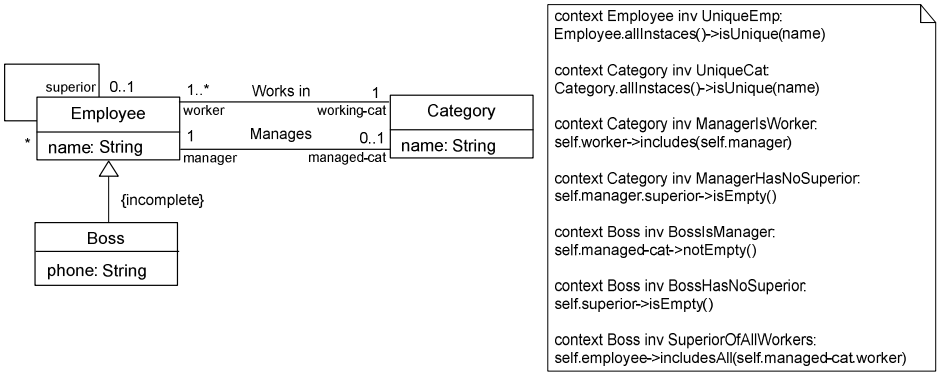

Fig. 2. UML class diagram and OCL integrity constraints for Employees and Categories

Given a structural schema such as the one in Figure 2, our method determines a number of properties, namely satisfiability, liveliness, constraint redundancy and state reachability, taking both the UML class diagram and the OCL constraints into account. This is achieved by means of two different steps.

First, we automatically translate the UML class diagram and the OCL integrity constraints into a logical representation. Classes and associations are represented by means of basic predicates. For instance, in the example of Figure 2, classes *Employee*

and *Category* become *Employee(e,name)* and *Category(c,name)*, and the association *WorksIn* is represented by the predicate *WorksIn(e,c)*.

The OCL constraints of the schema are represented in logic by means of conditions in denial form. For instance, the constraint *BossHasNoSuperior* will be translated into

$$\leftarrow Boss(e,phone) \wedge Superior(s,e)$$

The graphical constraints of the schema, such as cardinality and taxonomic constraints, also need to be translated into this kind of conditions.

A class diagram has a set of implicit constraints that need to be taken into account in the logical representation of the schema to preserve the semantics of the original one. For example, since UML is an object-oriented language, each instance has an internal object identifier (OID) which uniquely differentiates two instances, even though they are externally equivalent. Thus, additional constraints are needed in the logical representation to guarantee that two instances of the schema do not have the same OID. In the example of Figure 2 we need to specify the constraints:

$$\leftarrow Employee(e,name1) \wedge Employee(e,name2) \wedge name1 <> name2$$
$$\leftarrow Category(c,name1) \wedge Category(c,name2) \wedge name1 <> name2$$
$$\leftarrow Employee(x,name1) \wedge Category(x,name2)$$

These constraints guarantee that neither two instances of the same class nor two instances of different classes have the same OID.

As well as OIDs, the implicit constraints we can find in a class diagram are:

- In class hierarchies, an instance of a subclass must also be an instance of the superclass
- In associations or association classes, an instance of the association must link instances of the classes that define the association.
- In association classes, there cannot exist several instances linking exactly the same instances. Note that this is also true for associations without an association class, but an additional constraint is not needed in this case, since predicates representing n-ary (n>=2) associations have exactly n terms that can not be identical in two different instances of the predicate.

Once this translation is done, we are able to define the determination of each property as a constraint-satisfiability checking test. Then, we show how to use the CQC Method [7, 8] to determine those properties.

The CQC Method performs constraint-satisfiability checking tests by trying to construct a sample state satisfying a certain condition. The method uses different *Variable Instantiation Patterns (VIPs)* according to the syntactic properties of the conceptual schemas considered in each test.

For instance, to demonstrate that the class *Employee* of our example may have at least one instance, the CQC Method constructs the following state:

```
[manages(0, 1), category(1), worksIn(0, 1), employee(0)]
```

This means that it is possible to have an instance of *Employee* satisfying all the graphical, implicit and OCL constraints. The values of the instantiation correspond to a representative state of the information base. That is, this solution given by the CQC Method means that an instance of employee can exist if he or she works in a category (since all employees must work in a category) and is the manager of that category (since every category must have a manager and he or she must be one of its workers).

3. Translating a UML Structural Schema into Logic

In this section we propose a set of rules that, applied to a UML class diagram and a set of OCL constraints, result in a set of first-order formulas that represent the whole structural schema. We explain first how to obtain the formulas for the class diagram, taking into account its implicit and predefined constraints. Later, we propose a translation for user-defined OCL constraints. The complete logic representation of the schema can be found in the Appendix.

3.1. Translation of a UML Class Diagram

A UML class diagram is translated into a set of first-order formulas according to the following rules.

3.1.1. Translation of classes.

For each class C not being an association class, with a set of attributes $a_1, ..., a_n$, we define a base predicate C, with a term for each attribute. Additionally, since UML is an object-oriented language, there may exist several instances of a class externally identical, that is, with the same value in all their attributes. Thus, an additional term representing the internal object identifier (OID) must be included in the predicate C.

For example, the class *Employee* is translated into a predicate *Employee(e,name)*.

3.1.2. Translation of class hierarchies

Given a hierarchy with superclass $C_{sup,}$ denoted by the predicate $C_{sup}(c,asup_1,...,asup_n)$, and subclasses $C_{sub1},...,C_{subm}$ with attributes $\{asub1_1,...,asub1_p\}$, ...,$\{asubm_1,...,asubm_q\}$, we define a base predicate $C_{subi}(c,asubi_1,...,asubi_j)$ for each subclass.

For instance, the employee hierarchy with subclass *Boss* is translated into the predicate *Boss(e,phone)*, taking into account that the superclass *Employee* has already been translated into *Employee(e, name)*.

3.1.3. Translation of associations and association classes

Let R be an association between classes $C_1,...,C_n$. If R is not an association class, we define a base predicate $R(c_1,...,c_n)$. Otherwise, if R is an association class with attributes $a_1,...,a_m$, we define a base predicate $R(r,c_1,...,c_n,a_1,...,a_m)$. Although it is not strictly necessary, we also include an OID *r* so that all classes can be treated uniformly.

For example, the association *WorksIn* that relates *Employees* and *Categories* is translated into the predicate *WorksIn(e,c)*.

3.1.4. Translation of implicit and graphical constraints

First of all, we must guarantee that there cannot exist two instances with the same OID value. This implies defining two sets of rules, one of them to avoid having two literals of the same predicate with the same value in the OID, and the other to prevent the existence of two literals of different predicates with the same OID. Then, for each predicate $C(c,a_1,...a_n)$ we define the following constraint for each attribute a_i:

$\leftarrow C(c,a_1 1, \ldots, a_n 1) \wedge C(c,a_1 2, \ldots, a_n 2) \wedge a_i 1 <> a_i 2$

And for each pair of predicates we define the constraint:

$\leftarrow C_1(x,\ldots) \wedge C_2(x,\ldots)$

According to these rules, we must define the following constraints in our example:

$\leftarrow Employee(e,name1) \wedge Employee(e,name2) \wedge name1 <> name2$

$\leftarrow Employee(x,name1) \wedge Category(x,name2)$

Class hierarchies also require the definition of a set of constraints. Firstly, we must guarantee that each instance of a subclass C_{subi} is always an instance of the superclass. This is done by means of the rules:

$\leftarrow C_{subi}(c,asubi_1,\ldots,asubi_k) \wedge \neg IsC_{super}(c)$

$IsC_{super}(c) \leftarrow C_{super}(c,asuper_1,\ldots,asuper_n)$

In the example, the hierarchy of employees requires the following constraints:

$\leftarrow Boss(e,phone) \wedge \neg isEmp(e)$

$isEmp(e) \leftarrow Employee(e,name)$

Moreover, additional rules are sometimes required to guarantee that an instance of the superclass is not an instance of several subclasses simultaneously (*disjoint* constraint), and that an instance of the superclass is an instance of at least one of its subclasses (*complete* constraint). For each pair of subclasses C_{subi} , C_{subj} we define:

$\leftarrow C_{subi}(c,asubi_1,\ldots,asubi_k) \wedge C_{subj}(c,asubj_1,\ldots,asubj_l)$

and for each subclass C_{subi} of C_{super}:

$\leftarrow C_{super}(c,asuper_1,\ldots,asuper_n) \wedge \neg IsKindOfC_{super}(c)$

$IsKindOfC_{super}(c) \leftarrow C_{subi}(c,asubi_1,\ldots,asubi_m)$

Another set of constraints is needed for each association to guarantee the implicit constraint that an instance of the association can only relate existing instances of the classes defining the association. Then, for each association R, being or not an association class with OID r and attributes $r_1,\ldots,r_m$, represented by the predicate $R([r,],c_1,\ldots,c_n[,r_1,\ldots,r_m])$, for each c_i we define the constraint:

$\leftarrow R([r,],c_1,\ldots,c_n[,r_1,\ldots,r_m]) \wedge \neg IsC_i(c_i)$

$IsCi(c_i) \leftarrow C_i(c_i,x_1,\ldots,x_p)$

In our example, the association *WorksIn* requires the addition of the rules:

$\leftarrow WorksIn(e,c) \wedge \neg isEmp(e)$

$\leftarrow WorksIn(e,c) \wedge \neg isCat(c)$

$isEmp(e) \leftarrow Employee(e, name)$

$isCat(c) \leftarrow Category(c, name)$

Additionally, for the definition of association classes, we must guarantee that there are not several instances of an association class having the same value in the terms defining the instance. Then, if R is an association class, defined by classes $C_1,\ldots,C_m$, with attributes $a_1,\ldots a_m$, if there is no C_i with 1 as cardinality upper value in association R, we define the following constraint for each attribute a_i:

$\leftarrow R(r1,c_1\ldots c_m,a_1 1,\ldots,a_n 1) \wedge R(r2,c_1\ldots c_m,a_1 2,\ldots,a_n 2) \wedge a_i 1 <> a_i 2$

Finally, let *min..max* be a cardinality constraint attached to a class C_i in an association (class) R defined by classes $C_1,\ldots,C_n$. If *min*>0 the following constraint must be added:

$\leftarrow C_1(c_1,...) \wedge ... \wedge C_{i-1}(c_{i-1},...) \wedge C_{i+1}(c_{i+1},...) \wedge ... \wedge C_n(c_n,...) \wedge \neg MinR(c_1,...,c_{i-1},c_{i+1},...,c_n)$
$MinR(c_1,...,c_{i-1},c_{i+1},...,c_n) \leftarrow R([r1,]c_1,...,c_{i-1},c_i1,c_{i+1},...,c_n[,r_11,...r_m1]) \wedge ... \wedge$
$R([rmin,]c_1,...,c_{i-1},c_imin,c_{i+1},...,c_n[,r_1min,...,r_mmin]) \wedge c_i1 <> c_i2 \wedge ... \wedge c_i1 <> c_imin \wedge ...$
$\wedge c_imin\text{-}1 <> c_imin$

And if $max < *$, the following constraint is needed:
$\leftarrow R([r1,]c_1,...,c_{i-1},c_i1,c_{i+1},...,c_n[,r_11,...r_m1]) \wedge ... \wedge$
$R([rmax+1,]c_1,...,c_{i-1},c_imax+1,c_{i+1},...,c_n[,r_1max+1,...,r_mmax+1]) \wedge c_i1 <> c_i2 \wedge ... \wedge$
$c_i1 <> c_imax+1 \wedge ... \wedge c_imax <> c_imax+1$

As an example, we must define the following constraint to guarantee the lower multiplicity of class *Employee* in the association *WorksIn*:

$\leftarrow Category(c, name) \wedge \neg oneWorker(c)$
$oneWorker(c) \leftarrow worksIn(e,c)$

We also have to define the following one due to the upper multiplicity of Category:

$\leftarrow WorksIn(e, c1) \wedge WorksIn(e, c2) \wedge c1 <> c2$

3.2. Translation of OCL Integrity Constraints

We perform the translation of OCL integrity constraints into first-order logic in two steps. First, we transform each OCL expression into an equivalent one expressed in terms of the operations *select* and *size.* Once this has been done, we can apply a uniform treatment to all constraints in order to obtain the corresponding logic formulas.

3.2.1. Simplification of OCL operations

The first step in the translation process consists in the reduction of the number of OCL operations that appear in the constraints. Table 1 shows the OCL operations we consider, and gives their equivalent expressions in terms of operations *select* and *size.* These translations are iteratively applied until the only OCL operations that appear in the expression are *select* and *size.*

Table 1. Equivalences of OCL operations

Original expression	**Equivalent expression with *select* and *size***
source->includes(obj)	source->select(e | e=obj)->size()>0
source->excludes(obj)	source->select(e | e=obj)->size()=0
source->includesAll(c)	c->forall(e| source->includes(e))
source->excludesAll(c)	c->forall(e| source->excludes(e))
source->isEmpty()	source->size()=0
source->notEmpty()	source->size()>0
source->exists(e | body)	source->select(e | body)->size()>0
source->forall(e | body)	source->select(e | not body)->size()=0
source->isUnique(e | body)	source->select(e |source->select(e2 | e <>e2 and e2.body = e.body))->size()=0
source->one(e | body)	source->select(e | body)->size()=1
source->reject(e | body)	source->select(e | not body)

As an example we give the simplified form of the constraint *ManagerIsWorker* in the example of Figure 2:

context Category **inv** ManagerIsWorker**:**
self.worker->select(e |e = self.manager)-> size() > 0

Another example, where these equivalences have to be applied iteratively, is the constraint *SuperiorOfAllWorkers.* In a first step we simplify the *includesAll* operation:

context Boss **inv** SuperiorOfAllWorkers**:**
self.managed-cat.worker->forall(e|self.employee->includes(e))

Now we eliminate the *forall* operation:

context Boss **inv** SuperiorOfAllWorkers**:**
self.managed-cat.worker->select(e| self.employee->excludes(e))->size()=0

And, finally, the *excludes* operation:

context Boss **inv** SuperiorOfAllWorkers**:**
self.managed-cat.worker->select(e| not self.employee->select(e2 | e2=e)-> size=0)->size()=0

3.2.2. Translation of OCL invariants into logic

Once simplified, an OCL invariant has the following form:

context *C* **inv:** *path-exp*->select(e| body)->size() *opComp k*

where *C* is a class of the conceptual schema, *path-exp* is a sequence of navigations through associations, *opComp* is a comparison operator <, >, = or <> and *k* is an integer not lower than zero[1].

The translation of the simplified OCL invariants into logic depends on the specific operator after *size()*. We are going to see first how to translate the navigation defined by *path-exp* and the translation of the *select* operation. The select expression does not necessarily appear in the simplified OCL invariant, in which case it is not translated.

Tr-path(path-exp)

Let *path-exp* = $obj.r_1...r_n[.attr]$ be a path starting from an instance *obj* of a class *C*, or from a call to the *allInstances* operation on *C*, navigating through roles r_1 to r_n and, optionally, ending with the access to an attribute. Let *C(obj,...)* be the literal resulting from the translation of the class of which *obj* is an instance, and $R_i(obj_{i-1}, obj_i,...)$ be the literals corresponding to the association between roles r_{i-1} and r_i, and C_2 be the class where the attribute *attr* is defined. Then, this navigation path is translated into logic by means of the clause: $C(obj,...) \wedge R_1(obj,obj_1) \wedge ... \wedge R_n(obj_{n-1}, obj_n) \wedge C_2(obj_n,...)$.

For instance, the navigation *self.worker* appearing in constraint *ManagerIsWorker* will be translated into $Category(c,name) \wedge WorksIn(e,c)$.

Tr-select(e| body)

First we provide the translation of a select expression in its most simplified and usual form, where *body = path1 opComp path2*. In this case, the select operation is translated into:

[1] When <=k or >=k appear in the original invariant, they are translated into <k+1 and >k-1.

Tr-path(path1) $\wedge$ *Tr-path(path2)* $\wedge$ *obj1 opComp obj2*

where *obj1* and *obj2* are the objects obtained as a result of the navigation paths *path1* and *path2*, respectively. Note that if any of the paths is a constant or *e*, then it must not be translated.

For instance, the translation of the expression *select(e| e=self.manager)* appearing in the simplified OCL invariant of the constraint *ManagerIsWorker* will be translated into *Category(c,name)* $\wedge$ *Manages(e2,c)* $\wedge$ *e=e2*.

The body of a *select* operation can also contain, recursively, other *select* and *size* operations, that is, *body = path-exp*->select(e| body)->size() *opComp k*, where the select operation may not appear, and then it is not translated. We define the translation in terms of the translation of *path-exp* and the *select* operation as follows, depending on *opComp,* when *opComp* is <, > or =:

a) obj.r_1... r_{n-1}.r_n -> select(e| body)-> size() < k becomes

Tr-path(obj.r_1... r_{n-1}*)* $\wedge$ $\neg$*Aux(e,...,*e_m*,c)*

Aux(e,...,e_m*,c)*$\leftarrow$ *Tr-path*$_1$*(*r_n*)* $\wedge$ *Tr-select*$_1$*(e| body)*
$\wedge$ ... $\wedge$ *Tr-path*$_k$*(*r_n*)* $\wedge$ *Tr-select*$_k$*(e| body)*

b) obj.r_1... r_{n-1}.r_n -> select(e| body)-> size() > k becomes

Tr-path(obj.r_1... r_{n-1}*)* $\wedge$ *Tr-path*$_1$*(*r_n*)* $\wedge$ *Tr-select*$_1$*(e| body)*
$\wedge$ ... $\wedge$ *Tr-path*$_{k+1}$*(*r_n*)* $\wedge$ *Tr-select*$_{k+1}$*(e| body)*

c) obj.r_1... r_{n-1}.r_n -> select(e| body)-> size() = k becomes

Tr-path(obj.r_1... r_{n-1}*)* $\wedge$ *Tr-path*$_1$*(*r_n*)* $\wedge$ *Tr-select*$_1$*(e| body)*
$\wedge$ ... $\wedge$ *Tr-path*$_k$*(*r_n*)* $\wedge$ *Tr-select*$_k$*(e| body)* $\wedge$ $\neg$*Aux(e,...,*e_m*,c)*
Aux(e,...,e_m*,c)* $\leftarrow$ *Tr-path(obj.*r_1... r_{n-1}*)* $\wedge$ *Tr-path*$_1$*(*r_n*)* $\wedge$ *Tr-select*$_1$*(e| body)*
$\wedge$ ... $\wedge$ *Tr-path*$_{k+1}$*(*r_n*)* $\wedge$ *Tr-select*$_{k+1}$*(e| body)*

where the variables *e,...,*e_m needed by each Aux predicate correspond to the iteration variables of all the select operations in which the translated *body* is included, and *c* represents the contextual object.

Each translation Tr-path or Tr-select may be performed several times depending on the constant *k*. Each of the Tr-path$_i$ or Tr-select$_i$ expressions refers to the same translation but with different variables for those terms not coming from the translation of *obj.*r_1... r_{n-1}.

As an example, consider the OCL simplified invariant of constraint *SuperiorOfAllWorkers*:

context Boss **inv** SuperiorOfAllWorkers**:** self.managed-cat.worker->
select(e | self.employee->select(e2 | e2=e)->size()=0)->size()=0

applying the translation c) above we obtain, as a translation of the outer select:

Tr-path(self) $\wedge$ $\neg$*Aux(e,b)*
Aux(e,b)$\leftarrow$ *Tr-path(self)* $\wedge$ *Tr-path(employee)* $\wedge$ *Tr-select(e2| e2=e)*

and, after translating paths and selects, we get the following formulas:

$Boss(b,p) \wedge \neg Aux(e,b)$
$Aux(e,b) \leftarrow Boss(b,p) \wedge Superior(b, e2) \wedge e2=e$

Translation of an OCL invariant

Let $path\text{-}exp = obj.r_1 ... r_{n-1}.r_n$. Depending on each comparison operator, we define the translation of an OCL invariant in terms of the translation of the path expression (*Tr-path*) and the select operation (*Tr-select*) as follows:

a) **context** C **inv:** obj.r_1... $r_{n-1}.r_n$ -> select(e| body)-> size() < k becomes

$\leftarrow C(c,a_1,...,a_m) \wedge Tr\text{-}path(obj.r_1 ... r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$
$\wedge ... \wedge Tr\text{-}path_k(r_n) \wedge Tr\text{-}select_k(e|\ body)$

b) **context** C **inv:** obj.r_1... $r_{n-1}.r_n$ -> select(e| body)-> size() > k becomes

$\leftarrow C(c,a_1,...,a_m) \wedge \neg Aux(c)$

$Aux(c) \leftarrow Tr\text{-}path(obj.r_1 ... r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$
$\wedge ... \wedge Tr\text{-}path_{k+1}(r_n) \wedge Tr\text{-}select_{k+1}(e|\ body)$

c) **context** C **inv:** obj.r_1... $r_{n-1}.r_n$ -> select(e| body)-> size() = k becomes

$\leftarrow C(c,a_1,...,a_m) \wedge \neg Aux(c)$

$Aux(c) \leftarrow Tr\text{-}path(obj.r_1 ... r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$
$\wedge ... \wedge Tr\text{-}path_k(r_n) \wedge Tr\text{-}select_k(e|\ body)$

$\leftarrow C(c,a_1,...,a_m) \wedge Tr\text{-}path(obj.r_1 ... r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$
$\wedge ... \wedge Tr\text{-}path_{k+1}(r_n) \wedge Tr\text{-}select_{k+1}(e|\ body)$

d) **context** C **inv:** obj.r_1... $r_{n-1}.r_n$ -> select(e| body)-> size() <> k becomes

$\leftarrow C(c,a_1,...,a_m) \wedge Tr\text{-}path(obj.r_1 ... r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$
$\wedge ... \wedge Tr\text{-}path_k(r_n) \wedge Tr\text{-}select_k(e|\ body) \wedge \neg Aux(c)$

$Aux(c) \leftarrow Tr\text{-}path(obj.r_1 ... r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$
$\wedge ... \wedge Tr\text{-}path_{k+1}(r_n) \wedge Tr\text{-}select_{k+1}(e|\ body)$

Each translation Tr-path or Tr-select may be performed several times depending on the constant *k*. Each one of the Tr-path_i or Tr-select_i expressions refers to the same translation but with different variables for those terms not coming from the translation of $obj.r_1 ... r_{n-1}$. Clearly, the previous formalization becomes much simpler in the usual cases where the value of *k* is 0 or 1.

Intuitively, we may see that the translation of each OCL invariant defines a denial stating that a given situation cannot hold. The first part of each denial defines the logic representation of the path leading to the collection of instances to which the select and the size operations are applied. The second part, the one defined by the subindexes 1 to *k*, is required to guarantee that the elements that fulfill the select condition satisfy also the required comparison.

As an example, consider the OCL simplified invariant of constraint *ManagerIsWorker*:

context Category **inv:** self.worker -> select(e| e=self.manager)-> size() > 0

applying the translation b) above we obtain[2]:

$\leftarrow Category(c,name) \wedge \neg Aux(c)$
$Aux(c) \leftarrow Tr\text{-}path(self) \wedge Tr\text{-}path(worker) \wedge Tr\text{-}select(e|\ e=self.manager)$

and, after translating paths and selects, we get the following formulas which force all categories to have at least one worker who is also a manager.

$\leftarrow Category(c,name) \wedge \neg Aux(c)$
$Aux(c) \leftarrow Category(c,name) \wedge WorksIn(e,c) \wedge Manages(e2,c) \wedge e=e2$

To illustrate the translation of a complex OCL invariant, with *select* operation in the body of another *select*, we give the formulas for the simplified form of the constraint *SuperiorOfAllWorkers*:

context Boss **inv:** self.managed-cat.worker->
select(e | self.employee->select(e2 | e2=e)->size()=0)->size()=0

First, according to c) we obtain

$\leftarrow Boss(b,p) \wedge Tr\text{-}path(self) \wedge Tr\text{-}path(managed\text{-}cat) \wedge Tr\text{-}path(worker) \wedge$
Tr-select(e| self.employee->select(e2| e2=e)->size()=0)

Taking the translation of *Tr-select* obtained in the previous subsection we have the complete translation of this constraint, omitting repeated predicates:

$\leftarrow Boss(b,p) \wedge Manages(b,c) \wedge WorksIn(e,c) \wedge \neg Aux(e,b)$
$Aux(e,b) \leftarrow Boss(b,p) \wedge Superior(b, e2) \wedge e2=e$

It may also happen that the original expression does not include any OCL operation. Then the constraint has not been simplified and has the form:

context C **inv:** *path-exp opComp value*

where *value* is either a constant or another navigation path. The translation of these invariants into logic is:

$\leftarrow C(c, a_1,...,a_m) \wedge Tr\text{-}path(path\text{-}exp) \wedge Tr\text{-}path(value) \wedge obj1\ opComp\ obj2$

where *obj1* and *obj2* are the objects obtained as a result of the navigation path(s) *path-exp* and *value*. Note that if *value* is a constant then it must not be translated.

4. Reasoning on UML Structural Schemas using the CQC Method

4.1. The CQC Method in a Nutshell

The CQC Method performs query containment tests on deductive database schemas. Moreover, it is able to determine several properties on a database schema, namely state satisfiability, predicate liveliness, constraint redundancy and state reachability [7, 8]. It is a semidecidable procedure for finite satisfiability and unsatisfiability. This means that it always terminates when there exists a finite consistent state satisfying the property, or when the property is unsatisfiable (finitely or infinitely).

[2] Note that, since k=0, the translation of the select and the path must be performed only once.

Roughly, the CQC Method is aimed at constructing a state that fulfills a goal and satisfies all the constraints in the schema. As we will see, the goal to attain is formulated depending on the specific reasoning task to perform.

In this way, the CQC Method requires two main inputs besides the database schema definition itself. The first one is the definition of the *goal to attain*, which must be achieved on the database state that the method will try to obtain by constructing its information base. The second input is the set of *constraints to enforce*, which must not be violated by the constructed information base.

Then, in order to check if a certain property holds in a schema, this property has to be expressed in terms of an initial goal to attain (G_0) and the set of integrity constraints to enforce (F_0), and then ask the CQC Method Engine to construct a sample information base to prove that the initial goal G_0 is satisfied without violating any integrity constraint in F_0.

4.2. Using the CQC Method to Reason on UML and OCL Class Diagrams

In this subsection we show how to use the CQC Method in order to reason on UML class diagrams with OCL constraints. There are two kinds of reasoning tasks we may perform. The first ones consist in verifying, without the designer's intervention, that the schema satisfies a set of properties, namely satisfiability of the schema, liveliness of classes or associations and redundancy of constraints. On the other hand, state reachability requires the designer to ask questions and see if the answers given according to the CS correspond to what he expected.

We give a definition for each reasoning task, and which are the initial goal (G_0) and the set of constraints to enforce (F_0) in order to check it with the CQC Method. In most cases, F_0 coincides with the set of all the constraints of the schema (IC). The results of each reasoning task applied to the example of Figure 2 have been proved by means of a Prolog implementation of the method.

4.2.1. Satisfiability

A schema is satisfiable if there is a non-empty state of the information base in which all its integrity constraints are satisfied.

To check this property with the CQC Method, G_0 is to have any instance of any entity or relationship, and F_0 = IC. If the CQC Method engine succeeds in this task, i.e. if it finds a sample information base, then the schema is satisfiable. Otherwise, it is not.

The schema of Figure 2 is satisfiable, since it accepts at least a non-empty state that does not violate any integrity constraint. For instance, it may have an employee, which necessarily has to work in a category. Since each category must have a manager that is one of its workers, a valid sample state proving that the schema is satisfiable is:

{Employee(John), WorksIn(John,Sales), Category(Sales), Manages(John,Sales)}

Such a result is obtained by applying the CQC Method on the logical representation of the conceptual schema in Figure 2.

4.2.2. Liveliness of a class or association

Even if a schema is satisfiable, it may turn out that some class or association is empty in every valid state. Liveliness of classes or associations is another property that determines if a certain class or association can have at least one instance.

In this case, G_0 is to have any instance of the predicate representing the class or association to be checked, and F_0 = IC. If the CQC Method engine succeeds in this task then the class or association is lively.

Class *Category* of the schema in Figure 2 is lively, since there exists at least a state satisfying all the constraints in which *Category* has an instance. If we want to have a category, we also need at least one employee that works in it, and another one that is its manager. Besides, the manager must be one of the workers, so a valid state (obtained by the CQC Method) is:

{Category(Sales), WorksIn(John, Sales), Manages(John, Sales), Employee(John)}

At the same time, this state proves that class *Employee* and associations *Manages* and *WorksIn* are lively as well.

Let us see then if the association *Superior* and the class *Boss* are lively too. We have that there is at least a state in which *Superior* is not empty, which consists in an employee that works for another one, both of them working in the same category and the superior employee being the manager of the category. For example:

{Superior(Mary, John), Employee(Mary), Employee(John), WorksIn(Mary, Sales), WorksIn(John, Sales), Category(Sales), Manages(Mary, Sales)}

In contrast, if we reason on the liveliness of *Boss*, we see that to have an instance of *Boss* we need that he or she is the superior of all the employees that work in the category managed by that boss (constraint *SuperiorOfAllWorkers*). A state satisfying this condition would be one in which a boss does not manage any category, but this is prevented by constraint *BossIsManager*. Another way of satisfying this condition would be a state in which the category managed by the boss does not have workers, but constraint *ManagerIsWorker* forces each category to have at least a worker, its manager. Then, the only option is to have a boss that manages a category and that all the workers of that category (including the boss himself) are subordinates of that boss. But this is impossible according to the constraint *BossHasNoSuperior* and, therefore, the class *Boss* is not lively.

When eliminating either of these constraints, a state fulfilling the rest of conditions can be found and *Boss* becomes lively. For instance, if we remove the constraint *BossIsManager* we obtain the following state, in which the boss works in a category but is not a manager. Since every category must have a manager, it is another employee the one who manages the category:

{Boss(John), Employee(John), WorksIn(John, Sales), Category(Sales), Manages(Mary, Sales), WorksIn(Mary, Sales), Employee(Mary)}

All those results are obtained when applying the CQC Method on the logical representation of the conceptual schema in Figure 2.

4.2.3. Redundancy of an integrity constraint

An integrity constraint is redundant if integrity does not depend on it, that is, if the sates it does not allow are already not allowed by the rest of constraints.

Let Ic1 be one of the integrity constraints defined in the schema. In order to check if it is redundant, G_0 = Ic1, and F_0 = IC – {Ic1}. If the CQC Method engine is not capable of constructing such a state, then Ic1 is redundant.

If we analyze the constraints of the schema in Figure 2 we can see, for instance, that the constraint *BossHasNoSuperior* is redundant. We can try to build a state in which this constraint is violated while the rest are not, but it is not possible since if we want to have a boss, then he or she must be the manager of some category (constraint *BossIsManager*). Additionally, the constraint *ManagerHasNoSuperior* prevents the manager of every category from having a superior and thus, the constraint *BossHasNoSuperior* can never be violated.

There are other redundant constraints in this example, for instance between a graphical constraint and an OCL constraint. In particular, the constraint *ManagerIsWorker* implies that a category must have at least a worker, since it must have a manager and it must be one of its workers. At the same time, the cardinality constraint 1..* of *Employee* in the association *WorksIn* already forces the existence of at least a worker in each category. Both redundancies are confirmed by the execution of CQC Method with the corresponding parameters.

4.2.4. Reachability of a partially specified state

We may also be interested in more general properties of the schema, like checking whether certain states are accepted by the schema. This is usually known as checking reachability of partially specified states.

Let S be a partially specified state of the information base, that is, a set of instances probably inconsistent. If the CQC Method engine is capable of building a state with G_0 = S and F_0 = IC, then S is reachable.

We can do this in two ways, either by giving specific instances of some classes or associations, or by specifying a condition, that is, a state not fully instantiated.

For instance, taking the example in Figure 2, we may wonder whether it is possible to have a manager that is not a boss. Then we can try to construct a state in which an employee is the manager of a category and is not a boss. Assuming that the initial partially specified state is {Manages(John, Sales) $\wedge$ $\neg$Boss(John)}, a solution given by the CQC Method is:

{Manages(John, Sales), Employee(John), Category(Sales), WorksIn(John, Sales}

This means that the state is reachable, but as long as there exist the corresponding instances of *Employee* and *Category* and also the specified employee works in the category he manages.

Another question we could ask is whether there can be a boss who is not the superior of any employee, which may not be clear at first sight. If we try to construct a state satisfying {Boss(x) $\wedge$ $\neg$isSuperior(x)} with isSuperior(x) $\leftarrow$ Superior(x,e) we can see that this is not possible. The constraint *BossIsManager* forces a boss to be the manager of some category while the constraint *SuperiorOfAllWorkers* guarantees that all employees that work in the category managed by a boss are subordinates of that boss and, thus, the boss is the superior of all of them.

The previous result is also obtained by the CQC Method since it determines that there is no solution of the goal to attain in the logical representation of the conceptual schema in Figure 2.

5. Related Work

We review how reasoning on conceptual schemas has been addressed so far. We will start in ER conceptual schemas and later we will deal with UML conceptual schemas. As we will see, the main contribution of our approach is to deal with more expressive conceptual schemas than previous methods. We must state however that those methods are in general more efficient than ours for the particular cases they may handle.

5.1 Reasoning on ER Conceptual Schemas

The most popular task addressed in ER conceptual schemas is strong satisfiability checking, in particular regarding cardinality constraints. Strong satisfiability was introduced in [12] and their approach to determine this property consists in reducing the problem to solving a linear inequality system. This system is defined from the set of relationships and cardinality constraints of the schema. Then, a schema is strongly satisfiable if and only if there are solutions for its inequality system.

On the other hand, [10] determines strong satisfiability of a schema by means of a graph-theoretic approach. This work deals with int-cardinality constraints, which are more general than traditional ones since they allow gaps in the sets of permitted cardinalities.

The same method is used in [11], but this time it serves more specific purposes. Given a cardinality constraint set S, the method can find superfluous entities, i.e. entities whose population is empty in every instance of the schema satisfying S determine which is the minimal subset of constraints that causes a schema not to be satisfiable suggest strategies to resolve inconsistency

Another problem is approached in [5], which is the detection of potentially redundant associations in an ER schema. The method is based this time in adjacency matrixes, but it is incomplete since some types of redundancy involving more than one relationship between two entities cannot be detected.

Summarizing, we may see that several methods have approached reasoning on ER conceptual schemas, mainly through cardinality constraints. Nevertheless, none of them takes general-purpose integrity constraints into account, while we do.

5.2 Reasoning on UML Conceptual Schemas

Description Logics (DL) is a family of formalisms for knowledge representation, based on first-order logic [1]. In the last years, DL has gone beyond its traditional scope in Artificial Intelligence area to provide new alternatives and solutions to many topics in the database and conceptual modeling areas [3, 4, 6].

DL allows inferring represented knowledge from the knowledge explicitly contained in the knowledge base. Such an inference mechanism can be used to determine some properties of the structural schema, such as schema consistency (satisfiability), class consistency, class equivalence or class subsumption. DL assumes that the system should always check the desired properties in reasonable time and, thus, it uses to

restrict the expressive power of each specific DL to guarantee that the problem to be solved remains decidable.

An interesting approach to reasoning on UML specifications (i.e. class diagrams) is to translate them to DL and then use current standard DL-based reasoning systems on them [2] to automatically verify properties like the ones stated above. However, this approach does not deal with general-purpose OCL constraints since they may not be taken into account to guarantee decidability of the problem being handled.

An important exception on the treatment of OCL integrity constraints is the system USE [9], which allows to validate UML and OCL models by constructing snapshots representing system states with objects, attribute values and association links (something similar to our reachability of partially specified states). With this feature USE allows to validate whether the schema specifies the relevant knowledge of the domain, as perceived by the designer. Nevertheless, USE is not able to automatically verify whether the schema satisfies desirable properties like satisfiability, liveliness or redundancy.

Moreover, we see two important differences between USE and the work reported here. Firstly, in USE the designer must define a single operation by-hand to build each one of the states of the conceptual schema to be validated; while we allow defining them declaratively by stating (the subset of) the information they should contain.

Secondly, and most important one, in USE the generated snapshots are checked against the constraints and then rejected if some of them is violated. On the contrary, in our approach the partially specified states that violate some constraint are repaired by means of assuming additional information that allows repairing the violations. In this way, we obtain solutions that may not be generated in USE. For instance, in the example of the introduction, USE would conclude that Category(Marketing,8000) is not a valid snapshot, while we draw that it is possible to have such category if the conceptual schema contains also *Employee(Mary,9000)*, *WorksIn(Mary,Marketing)* and *RichEmp(Mary)*.

6. Conclusions and Further Work

We have proposed a new approach to reason on structural conceptual schemas specified in UML with OCL integrity constraints, both regarding the correctness of its structure and the states of the domain it accepts. In this sense, we have provided a set of automatic tests that can be performed on the schema, namely satisfiability, liveliness or redundancy, and also facilities in order to check if the schema represents the information expected by the designer.

Our approach consists of two main steps. First, we translate the UML class diagram and the OCL constraints into a first-order logic representation, and then we use the CQC Method, which performs constraint-satisfiability checking tests, in order to perform the reasoning and validation tasks stated above. The CQC Method is a semidecidable procedure for finite satisfiability and unsatisfiability. This means that it always terminates when there exists a finite consistent state satisfying the property, or when the property is unsatisfiable (finitely or infinitely).

We have illustrated the usefulness of our results by applying our approach to a simple conceptual schema. We have translated the whole schema into its logic

representation following our proposal and we have determined all properties pointed up in the paper by means of an implementation of the CQC Method.

The main contribution of our approach is to be able to deal with general-purpose OCL constraints. In particular, we may reason about OCL invariants defined by means of OCL operations that result in a boolean value plus *select* and *size* operations.

There are some interesting directions for further work to take from this point. First, we plan to provide an implementation of the first step of our method, that is, the translation of UML and OCL into logic. Also, we plan to extend the subset of the OCL language considered in order to improve the expressiveness of the constraints treated. Moreover, we would like also to extend the kind of reasoning we may perform on UML conceptual schemas by considering also its behavioral part.

References

[1] Baader, F., Calvanese, D., McGuiness, D., Nardi, D., Patel-Schneider, P., (Eds.): The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)

[2] Berardi, D., Calvanese, D., De Giacomo, G.: Reasoning on UML Class Diagrams. Artificial Intelligence 168(1-2) (2005) 70-118

[3] Borgida, A.: Description Logics in Data Management. IEEE Transactions on Knowledge and Data Engineering 7(5) (1995) 671-682

[4] Borgida, A., Lenzerini, M., Rosati, R.: Description Logics for Data Bases. In: F. Baader, D. Calvanese, D. McGuiness, D. Nardi, and P. Patel-Schneider, (eds.): The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003) 472-494

[5] Bowers, D. S.: Detection of Redundant Arcs in Entity Relationship Conceptual Models. ER 2003 Ws LNCS 2784 (2003) 275-287

[6] Calvanese, D., Lenzerini, M., Nardi, D.: Description Logics for Conceptual Data Modeling. In: J. Chomicki and G. Saake, (eds.): Logics for Databases and Information Systems. Kluwer (1998) 229-263

[7] Farré, C., Teniente, E., Urpí, T.: A New Approach for Checking Schema Validation Properties. In: Proc. 15th International Conference on Database and Expert Systems Applications (DEXA'04) (2004) 77-86

[8] Farré, C., Teniente, E., Urpí, T.: Checking Query Containment with the CQC Method. Data and Knowledge Engineering 53(2) (2005) 163-223

[9] Gogolla, M., Bohling, J., Richters, M.: Validation of UML and OCL Models by Automatic Snapshot Generation. «UML» 2003 LNCS 2863 (2003) 265-279

[10] Hartmann, S.: On the Consistency of Int-cardinality Constraints. 17th International Conference on Conceptual Modeling - ER'98 LNCS 1507 (1998) 150-163

[11] Hartmann, S.: Coping with Inconsistent Constraint Specifications. 20th International Conference on Conceptual Modeling - ER 2001 LNCS 2224 (2001) 241-255

[12] Lenzerini, M., Nobili, P.: On the Satisfiability of Dependency Constraints in Entity-Relationship Schemata. In: Proc. 13th International Conference on Very Large Databases - VLDB'87 (1987) 147-154

[13] Olivé, A.: Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. 17th Int. Conf. on Advanced Information Systems Engineering (CAISE'05) LNCS 3520 (2005) 1-15

[14] Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. 2nd edn. Addison-Wesley Professional (2003)

Appendix

We provide here the whole translation of the example in Figure 2, according to the method explained in Section 3.

Translation of classes and associations
Employee(e,name)
Category(c,name)
Boss(b,phone)
WorksIn(e,c)
Manages(e,c)
Superior(e1,e2)

Constraints for OIDs
←Employee(e, n1) ∧ Employee(e, n2) ∧ n1<>n2
←Category(c, n1) ∧ Category(c, n2) ∧ n1<>n2
←Employee(x, n1) ∧ Category(x, n2)

Constraints for hierarchies
←Boss(e,p) ∧ ¬isEmp(e)
isEmp(e) ← Employee(e,n)

Constraints for associations
←WorksIn(e,c) ∧ ¬isEmp(e,n)
←WorksIn(e,c) ∧ ¬isCat(c,n)
←Manages(e,c) ∧ ¬isEmp(e,n)
←Manages(e,c) ∧ ¬isCat(c,n)
←Superior(s,e) ∧ ¬isEmp(s,n)
←Superior(s,e) ∧ ¬isEmp(e,n)
isCat(c) ← Category(c,n)

Constraints for cardinalities
←Category(c,n) ∧ ¬oneEmp(c)
←Employee(e,n) ∧ ¬oneCat(e)
←Category(c,n) ∧ ¬oneMgr(c)
←WorksIn(e,c1) ∧ WorksIn(e,c2) ∧ c1<>c2
←Manages(e1,c) ∧ Manages(e2,c) ∧ e1<>e2
←Manages(e,c1) ∧ Manages(e,c2) ∧ c1<>c2
←Superior(s1,e) ∧ Superior(s2,e) ∧ s1<>s2
oneEmp(c) ← WorksIn(e,c)
oneCat(e) ← WorksIn(e,c)
oneMgr(c) ← Manages(e,c)

OCL constraints

UniqueEmp
context Employee **inv:**
Employee.allInstances()->select(e| Employee.allInstances()->select(e2 | e<>e2 and e2.name=e.name))->size()=0

←Employee(e,n) ∧ Employee(e2,n2) ∧ e<>e2 ∧ n2=n

UniqueCat
context Category **inv:**
Category.allInstances()->select(c| Category.allInstances()->select(c2 | c<>c2 and c2.name=c.name))->size()=0

←Category(c,n) ∧ Cateogory(c2,n2) ∧ c<>c2 ∧ n2=n

ManagerIsWorker
context Category **inv:** self.worker->select(e | e=self.manager)->size()>0

← Category(c,n) ∧ ¬Aux(c)
Aux1(c) ← Category(c,n) ∧ WorksIn(e,c) ∧ Manages(e2,c) ∧ e=e2

ManagerHasNoSuperior
context Category **inv:** self.manager.superior->size()=0

←Category(c,n) ∧ Manages(e,c) ∧ Superior(s,e)

BossIsManager
context Boss **inv:** self.managed-cat->size()>0

←Boss(e,p) ∧ ¬Aux2(e)
Aux2(e) ← Boss(e,p) ∧ Manages(e,c)

BossHasNoSuperior
context Boss **inv:** self.superior->size()=0

← Boss(e,p) ∧ Superior(s,e)

SuperiorOfAllWorkers
context Boss **inv:** self.managed-cat.worker->select(e | self.employee->select(e2 | e2=e)->size()=0)->size()=0

←Boss(b,p) ∧ Manages(b, c) ∧ WorksIn(e, c) ∧ ¬Aux3(e,b)
Aux3(e,b) ← Boss(b,p) ∧ Superior(b, e2) ∧ e2=e