Fixing up Non-executable Operations in UML/OCL Conceptual Schemas

Xavier Oriol[1], Ernest Teniente[1], and Albert Tort[2]

[1] Department of Service and Information System Engineering
Universitat Politècnica de Catalunya – BarcelonaTech
{xoriol,teniente}@essi.upc.edu
[2] Sogeti España, albert.tort@sogeti.com

Abstract. An operation is executable if there is at least one information base in which its preconditions hold and such that the new information base obtained from applying its postconditions satisfies all the integrity constraints. A non-executable operation is useless since it may never be applied. Therefore, identifying non-executable operations and fixing up their definition is a relevant task that should be performed as early as possible in software development. We address this problem in the paper by proposing an algorithm to automatically compute the missing effects in postconditions that would ensure the executability of the operation.

Keywords: Conceptual schema, UML, operations

1 Introduction

Pursuing the correctness of a conceptual schema is a key activity in software development since mistakes made during conceptual modeling are propagated throughout the whole development life cycle, thus affecting the quality of the final product. The high expressiveness of conceptual schemas requires adopting automated reasoning techniques to support the designer in this important task.

The conceptual schema includes both structural and behavioral knowledge. The structural part of the conceptual schema consists of a taxonomy of classes with their attributes, associations among classes, and integrity constraints which define conditions that the instances of the schema must satisfy [1].

The behavioral part of a conceptual schema contains all operations required by the system. Each operation is defined by means of a contract, which states the changes that occur on the Information Base (IB) when the operation is executed. In UML [2], an *operation contract* is specified by a set of *pre/postconditions*, which states conditions that must hold in the IB before/after the execution of the operation [3]. Such pre/postconditions are usually specified in OCL [4]. An operation is *executable* if there is at least one IB in which its preconditions are satisfied and such that the new IB obtained from applying its postconditions is consistent, i.e. satisfies all the integrity constraints. A non-executable operation is useless and the designer should avoid this situation by modifying its contract.

1.1 Motivation

Consider the class diagram in Fig. 1 stating information about medical teams, their expertise, physicians being members or managers of a team and their medical specializations. The OCL constraints provide additional semantics. *SpecialistOfTeamsExpertise* ensures that a physician is not a member of a medical team if he does not have its expertise. *ManagerIsMember* states that all managers of a medical team must also be members of that team. Finally, *ExclusiveMembership* states that members of a critical team can not be members of other teams. We assume that the attributes are primary keys of their owner classes.

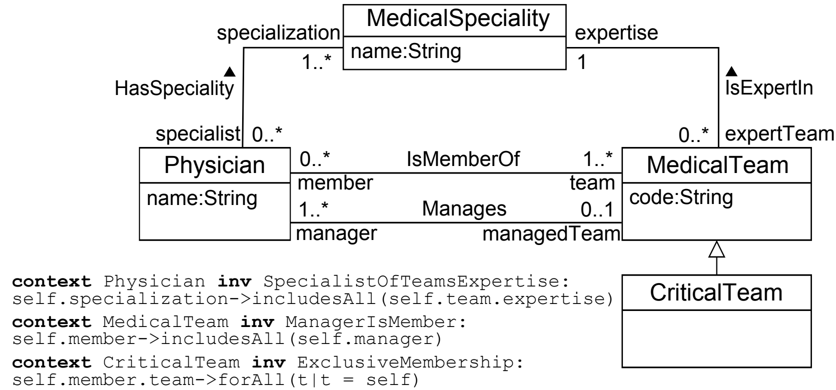

Fig. 1. A UML/OCL schema for the domain of medical teams

Consider now the following UML/OCL operation contracts aimed at inserting and deleting an instance of a *Critical Team*, respectively:

Operation: newCriticalTeam(p: Physician, s: MedicalSpeciality, cd: String)
pre: MedicalTeam.allInstances()->forAll(m|m.code<>cd) and p.specialization->includes(s) and p.managedTeam->isEmpty()
post: CriticalTeam.allInstances()->exists(c|c.oclIsNew() and c.code = cd and c.expertise = s and c.manager->includes(p))

Operation: deleteCriticalTeam(criticalTeam: CriticalTeam)
post: CriticalTeam.allInstances()->excludes(criticalTeam)
–we assume that deleting an instance of a class also deletes its links to other instances.

Both operations are non-executable. *newCriticalTeam* would always violate *ManagerIsMember* while *deleteCriticalTeam* would always violate the minimum cardinality 1 of the team role. This is because *ExclusiveMembership* forces all employees of a critical team to be members only of that team. So, those physicians will have no team when a critical team is deleted through this operation.

Several approaches have been proposed to identify non-executable operations [5,6,7,8,3,9] and most of them should be able to determine the non-executability of the previous operations. However, to our knowledge, none of them is able to provide the designer with additional information on how to modify the operation contracts to make them executable. Note that it is a very hard task to do it manually because of the huge expressive power of UML/OCL schemas.

1.2 Contribution

We propose a new method that allows identifying non-executable operations while providing information about how to fix up the problem. This information is given in terms of the missing effects on the operation postconditions that allow ensuring that all constraints are satisfied after executing the operation. In general, several different sets of missing effects for fixing up an operation may exist and the designer will have to decide which one to apply.

In our example, *newCriticalTeam* can be made executable by adding to its postcondition that the new manager is also a member of the team and removing all his/her previous memberships (to satisfy *ExclusiveMembership*). Regarding *deleteCriticalTeam*, there are several ways to make it executable. We could delete the physicians that were members of the critical team or, alternatively, we could add such physicians as members of other teams. Several additional effects might be considered depending on the IB to prevent a cascade violation of other constraints. All of them can be automatically computed in our approach.

Given an operation contract, our method starts generating a consistent IB that satisfies the precondition. This IB may optionally be manually modified by the designer. Then, the set of structural events required to satisfy the postcondition is computed. A structural event is a basic change in the IB, i.e. a insertion/deletion of an instance of a class or association. If the application of these structural events leads to the violation of an integrity constraint, our method applies a chase-like procedure to determine the additional structural events required to ensure the satisfaction of all the integrity constraints. This is achieved keeping the track of all the different minimal solutions that exist. We consider a solution to be minimal if no subset of the solution is itself a solution. From these results, the designer may know all the different alternatives (if any) that he/she can use to fix up the non-executability of the initial operation contract.

The contribution of this paper is threefold: (1) our method identifies non-executable operations while providing information to fix up this problem in the form of structural events that ensure operation executability when added in the postcondition, (2) our method can be used by current UML/OCL animation tools like USE [5] to find all the different ways to get a new consistent IB whenever a change applied to a previously consistent IB violates some integrity constraint, (3) we contribute to the conceptual-schema centric development grand challenge of *enforcement of integrity constraints in conceptual schemas* [10].

2 Basic Concepts and Notation

Information Base. An information system maintains a representation of the state of a domain in its *Information Base* (IB). The IB is the set of instances of the classes and associations defined in the conceptual schema. The integrity constraints of the conceptual schema define conditions that the IB must satisfy. We say that an IB is consistent if no constraint is violated on it.

Logic Formalization of the Schema. As proposed in [11] we formalize each class C in the schema with attributes $\{A_1, \ldots, A_n\}$ by means of a base

atom $c(Oid, A_1, \ldots, A_n)$, and each association R between classes $\{C_1, \ldots, C_k\}$ by means of a base atom $r(C_1, \ldots, C_k)$. The set of instances of the IB is represented by the set of facts about the atoms obtained from such formalization. An atom $v(\overline{x})$ is derived (i.e., a view) if it is defined by a rule of the form: $v(\overline{x}) \leftarrow l_1(\overline{x_1}), \ldots, l_k(\overline{x_k})$ where the variables in $\overline{x}$ are taken from $\overline{x_1}, \ldots, \overline{x_k}$. Each literal l_i is an atom, either positive or negative. Every variable occurring in the head or in a negative atom of the body must also occur in a positive atom of the body.

Structural Events. A *structural event* is an elementary change in the population of a class or association [1]. I.e. a change in the IB. We consider four kinds of structural events: class instance insertion, class instance deletion, association instance insertion and association instance deletion. We denote insertions by ι and deletions by δ. Given a base atom $P(\overline{x})$, where $\overline{x}$ stands for the set of variables $x_1, \ldots, x_n$, insertion structural events are formally defined by the formula $\forall \overline{x}(\iota P(\overline{x}) \leftrightarrow P^n(\overline{x}) \wedge \neg P(\overline{x}))$, while deletion structural events by $\forall \overline{x}(\delta P(\overline{x}) \leftrightarrow P(\overline{x}) \wedge \neg P^n(\overline{x}))$, where P^n stands for predicate P evaluated in the new IB, i.e. the one obtained after applying the change.

Dependencies. A *Tuple-Generating Dependency (TGD)* is a formula of the form $\forall \overline{x}, \overline{z}\big(\varphi(\overline{x}, \overline{z}) \rightarrow \exists \overline{y} \psi(\overline{x}, \overline{y})\big)$, where $\varphi(\overline{x}, \overline{z})$ is a conjunction of base literals (i.e. positive or negative atoms) and built-in literals (i.e. arithmetic comparisons) and $\psi(\overline{x}, \overline{y})$ is a conjunction of base atoms. A *denial constraint* is a special type of TGD of the form $\forall \overline{x}\big(\varphi(\overline{x}) \rightarrow \bot\big)$, in which the conclusion only contains the $\bot$ atom, which cannot be made true.

A *Disjunctive Embedded Dependency (DED)* is a TGD where the conclusion, i.e. $\psi(\overline{x}, \overline{y})$, is a disjunction of base atoms. A *Repair-Generating Dependency (RGD)* is a DED where the premise, i.e. $\varphi(\overline{x}, \overline{z})$, contains necessarily at least one structural event and optionally a derived negative atom, whereas the conclusion is either a single structural event or a disjunction of several structural events, i.e. it has the form $Ev_1 \vee \ldots \vee Ev_k$, where each Ev_i is a structural event. An *Event-Dependency Constraint (EDC)* is an RGD in which the conclusion only contains the atom $\bot$.

3 Determining the Missing Effects of Postconditions

Given a UML/OCL structural schema and an operation Op to be analyzed, our goal is to determine whether Op is executable and to provide information to fix up the problem if this is not the case. Our method starts by automatically generating a consistent IB satisfying the operation precondition to test whether Op is executable in such IB. The designer could also define his preferred initial IB from scratch or by modifying the automatically generated one.

Then, our method translates the postcondition of such operation into a set of structural events $EV = (Ev_1, \ldots, Ev_k)$. If the IB resulting from applying EV to the initial state is consistent, then Op is executable. Otherwise, our method looks for additional *repairing* structural events, $RE = (Re_1 \ldots, Re_m)$, such that we get a consistent IB when applying $EV \cup RE$. We want to keep the set RE

minimal in the sense that there is no $RE' \subsetneq RE$ such that $EV \cup RE'$ leads also to a consistent IB. Note that, in particular, Op is executable if RE is an empty set of structural events.

In general, several repairs RE_i may exist since there may be different ways of satisfying an integrity constraint. It may also happen that no RE is obtained. That means that EV cannot be applied to the IB without necessarily violating any constraint. I.e., there is no way to fix up the executability of the operation by just considering additional effects in the postcondition.

Our method computes the different repairing sets RE_i by means of the following steps: (1) encoding the UML/OCL conceptual schema into logic, (2) obtaining a set of rules, the repair-generating dependencies, that allows identifying when a constraint is violated and computing the structural events for repairing such violation, and (3) chasing the repair-generating dependencies to obtain the repairing sets RE_i.

3.1 Encoding the UML/OCL Conceptual Schema into Logic

We must encode first the UML/OCL conceptual schema into logic as proposed in [11]. Recall that each class C in the schema with attributes $\{A_1, \ldots, A_n\}$ is encoded as $c(Oid, A_1, \ldots, A_n)$, each association R between classes $\{C_1, \ldots, C_k\}$ is encoded as $r(C_1, \ldots, C_k)$ and each association class R between $\{C_1, \ldots, C_k\}$ and with the attributes $\{A_1, \ldots, A_n\}$ as $r(R, C_1, \ldots, C_k, A_1, \ldots, A_n)$. Without loss of generality, we will use the primary key attributes of classes as their *oid*.

As an example, the schema in Fig. 1 would be encoded as follows:

physician(P), *medicalSpeciality(MS)*, *hasSpeciality(P, MS)*, *medicalTeam(T)*
isExpertIn(T,MS), *isMemberOf(P,T)*, *manages(P,T)*, *criticalTeam(T)*

Each UML/OCL integrity constraint is encoded as a denial constraint as proposed in [11]. For example, the encoding of the first two OCL constraints in our running example is the following:

$$manages(P,T) \wedge \neg isMemberOf(P,T) \rightarrow \bot \quad (1)$$
$$isMemberOf(P,T) \wedge isExpertIn(T,MS) \wedge \neg hasSpeciality(P,MS) \rightarrow \bot \quad (2)$$

Rule 1 states that there may not be a medical team T managed by a physician P who is not a member of T, while rule 2 prevents a physician being a member of a medical team if his/her specializations do not include the expertise of the team.

To ensure that denial constraints are defined only in terms of base predicates, we assume that each OCL integrity constraint C has the form `context C inv: ExpBool`, where `ExpBool` is defined according to the following syntax rules (where `OpComp` is any OCL comparison operator):

```
ExpBool    ::= ExpBool ∧ ExpBool        | ExpBool ∨ ExpBool
               | ExpOp
ExpOp      ::= Path->excludesAll(Path) | Var.Member->includesAll(Path)
               | Path->excludes(Path)   | Var.Member->includes(Var)
               | Path->isEmpty()        | Path->forAll(Var| ExpBool)
               | Path OpComp Constant   | not Path.oclIsKindOf(Class)
               | Path OpComp Path       | Path.oclIsKindOf(Class)
Path       ::= Var.Navigation           | Class.allInstances().Navigation
Navigation ::= Member.Navigation        | oclAsType(Class).Navigation
               | Member                 | Attribute
               | oclAsType(Class)
```

We also encode into logic the graphical and structural constraints of the UML schema, i.e. primary key constraints, referential integrity constraints, identifiers of association classes, disjointness and completeness integrity constraints and maximum cardinality constraints (see [11] for the details of this encoding).

Assuming that denial constraints are defined only in terms of base predicates is not a restrictive assumption since the constraints we can handle are a superset of those constraints specified according to the patterns defined in [12], which have been shown to be useful for defining around the 60% of the integrity constraints found in real schemas. The only exception is the path inclusion constraint pattern for which we can only specify the situations that are compliant to our grammar.

3.2 Obtaining Repair-Generating Dependencies

The next step is to obtain the repair-generating dependencies (RGDs) that will allow us to identify the situations where an integrity constraint is violated by the current set of structural events under consideration and also to compute the sets of structural events RE_i which ensure that applying $EV \cup RE_i$ to the initial IB leads to a new consistent IB.

We start by describing the transformation required by general UML/OCL constraints, i.e. those that have been encoded into logic. Then, we show how to handle minimum cardinality constraints.

Dependencies For General UML/OCL Constraints. The RGDs for a general UML/OCL constraint *ic* are obtained in two steps. First, we generate the Event-Dependency Constraints (EDCs) for *ic*. Then, for each EDC we obtain a corresponding RGD whenever possible.

Generating Event-Dependency Constraints. Each denial constraint obtained as a result of the logic encoding of the UML/OCL constraint will be translated into several dependencies. Each such dependency will prevent a different situation in which the constraint would be violated in the new IB. This is achieved by replacing each literal in the denial by the expression that allows us to compute it in the new IB. Positive and negative literals must be handled differently according to the following formulas:

$$\forall \overline{x}(P^n(\overline{x}) \leftrightarrow (\iota P(\overline{x}) \wedge \neg P(\overline{x})) \vee (\neg \delta P(\overline{x}) \wedge P(\overline{x}))) \quad (3)$$

$$\forall \overline{x}(\neg P^n(\overline{x}) \leftrightarrow (\neg \iota P(\overline{x}) \wedge \neg P(\overline{x})) \vee (\delta P(\overline{x}) \wedge P(\overline{x}))) \quad (4)$$

Rule 3 states that an atom $P(\overline{x})$ (e.g. $medicalTeam(neurology)$) will be true in the new IB if it was false in the old IB but its insertion structural event has been applied (e.g. the medical team *neurology* did not exist in the previous IB but it has been inserted right now) or if it was already true in the old IB and its deletion structural event has not been applied (e.g. the medical team *neurology* existed in the previous IB and it has not been removed). In an analogous way, rule 4 states that $P(\overline{x})$ will be false in the new IB if it was already false and it has not been inserted or if it has been deleted.

Algorithm 1 getEventDependencies($premise \rightarrow \bot$)

```
EDC := {∅ → ⊥}
for all Literal P in premise do
  EDC_Pre := EDC
  EDC := ∅
  for all Dependency premise_Pre → ⊥ in EDC_Pre do
    if P is Built-in-literal then
      EDC := EDC ∪ {premise_Pre ∧ P → ⊥ }
    else
      if P is positive then
        EDC := EDC ∪ {premise_Pre ∧ P ∧ ¬δP → ⊥} ∪ {premise_Pre ∧ ιP ∧ ¬P → ⊥}
      else
        EDC := EDC ∪ {premise_Pre ∧ P ∧ δP → ⊥} ∪ {premise_Pre ∧ ¬ιP ∧ ¬P → ⊥}
      end if
    end if
  end for
end for
EDC.removeFirst()
return EDC
```

By applying the substitutions above, we get a set of EDCs that state all possible ways to violate a constraint by means of the structural events of the schema. EDCs are grounded on the idea of *insertion event rules* which were defined in [13] to perform integrity checking in deductive databases. In general, we will get $2^k - 1$ EDCs for each denial constraint dc, where k is the number of literals in dc. The pseudocode of the algorithm *getEventDependencies*, which performs this transformation, is shown in Algorithm 1.

Intuitively, the algorithm interprets each literal P as P^n and performs an unfolding according to the definition given by formulas 3 and 4. The first dependency generated corresponds to a dependency that would be activated just in case the constraint was violated in the previous IB. Taking advantage of the guaranteed consistency of the initial IB generated by our method, we can safely delete such dependency.

Applying Algorithm 1 to the constraint *ManagerIsMember*, which has been encoded into the denial constraint $manages(P,T) \wedge \neg isMemberOf(P,T) \rightarrow \bot$, we get the following event-dependency constraints:

$$manages(P,T) \wedge \neg\delta manages(P,T) \wedge isMemberOf(P,T) \wedge \delta isMemberOf(P,T) \rightarrow \bot \quad (5)$$

$$\neg manages(P,T) \wedge \iota manages(P,T) \wedge \neg isMemberOf(P,T) \wedge \neg\iota isMemberOf(P,T) \rightarrow \bot \quad (6)$$

$$\neg manages(P,T) \wedge \iota manages(P,T) \wedge isMemberOf(P,T) \wedge \delta isMemberOf(P,T) \rightarrow \bot \quad (7)$$

Rule 5 is an EDC stating that the constraint will be violated for a physician p and a team t if we delete the fact that p is a member of t but we do not delete at the same time that p is a manager of t. EDC 6 identifies a violation to happen when a new manager p of t is inserted without inserting p as a member of t at the same time. EDC 7 states that *ManagerIsMember* will be violated if we delete the membership association among p and t while inserting the manage association.

Obtaining Repair-Generating Dependencies (RGDs). EDCs let us identifying the situations where an integrity constraint is violated as a consequence of the application of a set of structural events. However, they do not directly provide any information on how this violation could be repaired by considering additional structural events. We transform EDCs into RGDs for this purpose by means of the algorithm *getRepairDependencies*, reported in Algorithm 2.

Intuitively, each negated structural event in the premise of the dependency constraint represents a different way to repair the constraint. Therefore, the negated structural events of the constraint are removed from the premise and placed positively in the conclusion. If there is more than one negated structural event in the premise, the conclusion of the RGD will be a disjunction of structural events. Note that we will obtain exactly one RGD for each EDC.

Applying Algorithm 2 to the EDCs defined by the rules 5, 6, 7, we get the following RGDs:

$$manages(P,T) \wedge isMemberOf(P,T) \wedge \delta isMemberOf(P,T) \rightarrow \delta manages(P,T) \quad (8)$$

$$\neg manages(P,T) \wedge \iota manages(P,T) \wedge \neg isMemberOf(P,T) \rightarrow \iota isMemberOf(P,T) \quad (9)$$

$$\neg manages(P,T) \wedge \iota manages(P,T) \wedge isMemberOf(P,T) \wedge \delta isMemberOf(P,T) \rightarrow \bot \quad (10)$$

Rule 8 is an RGD stating that when the structural event $\delta isMemberOf(p,t)$ occurs in an IB where p is a manager of t, then it is also required the structural event $\delta manages(p,t)$ to take place in order ensure that the constraint *ManagerIsMember* will not be violated. In a similar way, RGD 9 establishes that $\iota manages(p,t)$ requires $\iota isMemberOf(p,t)$ to take place as well. Rule 10 is exactly the EDC 7 meaning that no RGD can be obtained from it. That is, there is no possible way to repair the situation identified by EDC 7. In other words, structural events $\iota manages(p,t)$ and $\delta isMemberOf(p,t)$ cannot happen together.

Algorithm 2 getRepairDependencies($premise \rightarrow \bot$)

```
new_Conclusion := ⊥
new_Premise := ⊤
for all Literal P in premise do
    if P is negated structural event then
        new_Conclusion := new_Conclusion ∨ positive(P)
    else
        new_Premise := new_Premise ∧ P
    end if
end for
return new_Premise → new_Conclusion
```

Dependencies for Minimum Cardinality Constraints. Repair-generating dependencies for min. cardinality constraints can be directly generated by taking advantage of the precise semantics of this constraint. The following rules summarize how to obtain such RGDs for a minimum cardinality constraint of 1 in a binary association R between members C_1, C_2:

$$c_1(Oid_1, ...) \wedge \neg someRelationAlife(Oid_1) \rightarrow \delta c_1(Oid_1, ...) \vee \iota r(Oid_1, Oid_2) \quad (11)$$
$$someRelationAlife(Oid_1) \leftarrow r(Oid_1, Oid_2) \wedge \neg\delta r(Oid_1, Oid_2)$$
$$\neg c_1(Oid_1, ...) \wedge \iota c_1(Oid_1, ...) \rightarrow \iota r(Oid_1, Oid_2) \quad (12)$$

Applying such patterns to the minimum cardinality constraint on the role *manager* in the example of Figure 1, we would get the following RGDs:

$$medicalTeam(T) \wedge \neg someManagerAlife(T) \rightarrow \delta medicalTeam(T) \vee \iota manages(P, T) \quad (13)$$
$$someManagerAlife(T) \leftarrow manages(P, T) \wedge \neg\delta manages(P, T)$$
$$\neg medicalTeam(T) \wedge \iota medicalTeam(T) \rightarrow \iota manages(P, T) \quad (14)$$

RGD 13 states that if the IB contains a medical team for which all its manages associations have been deleted, then either we delete the medical team or we insert a new manages association in order to satisfy the minimum cardinality constraint. We know whether it has at least one manage association which have not been deleted by means of the derived atom *someManagerAlife*. RGD 14 asserts that whenever a medical team is created a manage association for this team must be created as well.

3.3 Chasing Repair-Generating Dependencies

Once we have the $RGDs$, we need an initial information base IB and the initial structural events EV in order to compute which are the missing structural event sets RE_i such that $EV \cup RE_i$ leads the current IB to a new consistent IB. We first explain how to obtain such initial IB and EV and then, how do we chase the $RGDs$ using IB and EV to compute the different RE_i.

Obtaining the initial IB and structural events. We need a consistent IB compliant with the precondition of the operation to test. There exist several proposals that allow obtaining such IB automatically from an OCL precondition. Most of these methods are based on translating the schema in some logic formalism and check for a witness of the satisfiability of the precondition. We can use any of them for our purposes. In our example, by applying [8] to the precondition of $newCriticalTeam$ we would get:

$$\begin{array}{ll} medicalTeam(t1) & medicalSpeciality(neurology) \\ isExpertIn(t1, neurology) & manages(mary, t1) \\ physician(john) & physician(mary) \\ isMemberOf(john, t1) & isMemberOf(mary, t1) \\ hasSpeciality(john, neurology) & hasSpeciality(mary, neurology) \end{array}$$

Table 1. Summarized mapping from OCL to structural events

OclExpression	Generated Structural Events
Class.allInstances()->exists(x|x.oclIsNew() ...)	$\iota Class(x, ...)$ and superclasses
Class.allInstances()->excludes(x)	$\delta Class(x, ...)$ and super/subclasses
x.oclIsTypeOf(Class)	$\iota Class(x, ...)$ and superclasses
not(x.oclIsTypeOf(Class))	$\delta Class(x, ...)$ and subclasses
x.memberEnd->includes(y)	$\iota Association(x, y)$
x.memberEnd->excludes(y)	$\delta Association(x, y)$

Now, to obtain the structural events EV we use the mapping from OCL postcondition expressions to structural events that was initially proposed in [8], and which is briefly summarized in Table 1.

Where any $\delta Class(x, ...)$ is followed by several $\delta Association(x, y)$ corresponding to the association links of x as a member of Class.

By applying this mapping to the postcondition of $newCriticalTeam$ in the previously shown IB, we get the structural events:

$$EV = \{\iota criticalTeam(t2), \iota medicalTeam(t2), \iota isExpertIn(t2, neurology), \iota manages(john, t2)\}$$

Chasing RGDs to compute repairing structural events. We must now chase the RGDs to determine the additional repairing sets of structural events RE_i that make the application of $EV \cup RE_i$ to the initial IB leading to a consistent IB′. Intuitively, an RGD is chased by querying its premise on the initial IB and the set of structural events under consideration (i.e. EV and the subset of RE_i already determined). Then, for each set of constants satisfying this query, one of the structural events in the conclusion must belong to $EV \cup RE_i$ to ensure that the constraint from which we have obtained the RGD is not violated. Disjunctions in the conclusion correspond to alternative solutions that keep the IB consistent. Existential variables in the conclusion are handled either by considering an existing constant in $IB \cup EV \cup RE_i$ or by inventing a new one (VIPs approach [14]). They also define different possible ways of repairing the constraint. This chasing process is formalized in Algorithm 3.

Algorithm 3 chaseRGDs($RGDs$, IB, EV, RE, $Result$)

```
D := getViolatedDependency(RGDs, IB, EV, RE)
if D = null then
   Result.add(RE)
else
   for all Literal R in (D.conclusion) do
      σ_rs := getRepairingSubstitutions(R, IB, EV, RE)
      for all σ_r in σ_rs do
         chaseRGDs(RGDs, IB, EV, RE ∪ {Rσ_r}, Result)
      end for
   end for
end if
```

Initially, the algorithm is called with $RE = \emptyset$ and $Result = \emptyset$. The *getViolatedDependency* function looks for a dependency being violated and returns it substituting its variables for the constants that provoke the violation. If no dependency is violated, then $EV \cup RE$ is already executable.

To repair the dependency we try all the possible literals in its conclusion. Moreover, for each of these literals, we try all the suitable variable-to-constant substitutions for the existential variables of the literal. This is achieved by means of the *getRepairingSubstitutions* function which implements the VIPs approach, thus, returning as many different substitutions as different constants may take each variable according to the currently used constants in $IB \cup EV \cup RE$.

Once the dependency is repaired, we continue looking for other/new violated dependencies by means of a recursive call to the same algorithm.

We illustrate this execution by applying the operation $newCriticalTeam$, using the previously obtained IB and EV. The relevant RGDs to facilitate understanding of the example are the following:

$$\neg medicalTeam(T) \land \iota medicalTeam(T) \rightarrow \iota manages(P,T) \quad (15)$$

$$\neg manages(P,T) \land \iota manages(P,T) \land \neg isMemberOf(P,T) \rightarrow \iota isMemberOf(P,T) \quad (16)$$

$$\neg criticalTeam(C) \land \iota criticalTeam(C) \land \neg isMemberOf(P,C) \land \iota isMemberOf(P,C) \land isMemberOf(P,T) \land T <> C \rightarrow \delta isMemberOf(P,T) \quad (17)$$

The algorithm starts looking for a violated RGD. Although the premise of 15 holds, this RGD is not violated because its conclusion is already included in EV. Thus, the algorithm picks 16. Indeed, the premise of 16 evaluates to true because of the literal $\iota manages(john,t2)$. Therefore, its conclusion $\iota isMemberOf(john,t2)$ must be included in RE. In this way, the method repairs a violation of the *ManagerIsMember* constraint.

Now, because of this new literal in RE, the premise of 17 holds. Indeed, the literals $\iota CriticalTeam(t2)$, $\iota isMemberOf(john,t2)$, $isMemberOf(john,t1)$ produce a violation. For repairing it, we add the conclusion $\delta isMemberOf(john,t1)$ to RE. In this case, we have repaired a violation of the *ExclusiveMembership* constraint that was produced when repairing *ManagerIsMember*.

Finally, no more RGDs are violated, so, the result consists of just one RE:

$$RE = \{\iota isMemberOf(john,t2), \delta isMemberOf(john,t1)\}$$

From this result, the designer may realize that the postcondition of *newCriticalTeam* is underspecified since it does not state that the new manager of the team must also be added as a member of the team and that the old membership of this manager must be deleted. These additional effects are required to satisfy the *ManagerIsMember* and the *ExclusiveMembership* constraints.

It is worth noting that this kind of feedback is very relevant for the designer to ensure that all the operations of the schema are executable since in general it is very hard to manually identify how to fix the non-executability of an operation.

4 Experiments

We have implemented a prototype tool of our approach to show the scalability of our method in real conceptual schemas. The analysis of an operation in our tool is performed as follows: (1) loading a conceptual schema from an XMI file, (2) selecting the operation to analyze, (3) optionally modifying the automatically generated initial IB in which to apply the operation, (4) determining

the sets of minimal structural events RE_i that, when applied together with the postcondition, bring the current IB to a new consistent IB.

We have applied this tool to help us define correct operations in the DBLP case study [15], whose structural conceptual schema has 17 classes, 9 specialization hierarchies, 18 associations and 25 OCL integrity constraints. When translated to our logical formalization this schema amounts to 128 denial constraints.

We have defined 11 different operations aimed at adding or removing publications from the schema (like new journal paper, or new edited book) and we have checked whether they were executable. These experiments have been performed with a C# implementation of the reasoning method on an Intel Core i7-4700HQ 2.4GHz processor, 8GB of RAM with Windows 8.1 and the average execution time has been about 50-55 seconds.

We have repeated the experiments by removing randomly some statement in the postcondition of each operation and checked the time required to compute the missing structural events. We have found that our tool has been able to recompute exactly the events corresponding to the missing statements with a similar amount of time as before.

We summarize our results in Table 2 by showing the results for those operations having a larger number of structural events in the postcondition. The first column states the name of the operation. The second column gives the number of instances in the initial IB satisfying the precondition of each operation and used by the test. The third column shows the number of structural events in the postcondition. The last three columns show, respectively, the time (in seconds) required to check executability of the operation and to compute the missing structural events when 1 or 2 statements were removed from the postcondition.

Table 2. Execution time for some of the operations in DBLP

Operation	Initial IB	Struct. events	Time	1 Miss	2 Miss
newAuthoredBook	6	5	50.72s	50.86s	52.21s
newEditedBook	5	6	50.91s	85.21s	52,80s
newBookSeriesIssue	6	9	51.74s	59.87s	52.64s
newJournalPaper	14	5	51.23s	51.68s	52.61s
delAuthoredBook	11	5	52.31s	50.64s	51.39s
delBookchapter	10	4	52.96s	50.58s	50.97s
delBookSeriesIssue	16	8	52.23s	50.92s	51.11s
delEditedBook	11	6	50.92s	50.86s	51.11s
delJournalPaper	17	5	51.23s	51.47s	51.34s
		Average	51.58s	55.79s	51.80s

5 Related Work

Previous proposals can be classified according to the following approaches: (1) checking desirable properties of operations, (2) animating the operations to explore their behavior, and (3) automatically generating operations contracts.

Checking desirable properties of operations. Several techniques have been proposed to check desirable properties of an OCL operation, such as executabil-

ity [6,8,3,9]. These techniques are fully automatic and aimed at obtaining a consistent IB that proves the property being checked, i.e. an IB where the operation can be executed when checking for executability. However, if no such IB is obtained, no feedback is provided to the designer to help him/her fix up the operation definition. This is, in fact, its main drawback as compared to ours.

Operation animation. Some proposals analyze the operation execution by means of animation [5,7,16]. Animation is achieved by simulating the execution of an operation in a specific IB and checking whether this execution violates some integrity constraint [5,7] or by reducing the satisfaction of the postcondition and class invariants to SAT [16]. Similarly to the previous approach, no feedback is provided to the designer to allow him/her to find the additional structural events required not to violate any constraint. As we have seen, having to do this manually is time consuming and error prone due to the difficulty of having to analyze by hand all the interactions among possible violations of the constraints.

Automatic generation of operation contracts. [17] addresses the automatic generation of basic operation contracts from the structural part of the conceptual schema which are ensured to be executable regarding the constraints of the class diagram and some simple provided stereotypes. However, this proposal is not able to deal with general OCL constraints nor with user-defined domain events as we do in this paper.

Summarizing, we may conclude that the approach we present in this paper is the first one that, given an operation contract and an IB, is able to automatically compute all missing structural events that should be covered by the postcondition to make the operation executable.

6 Conclusions

Ensuring the quality of a conceptual schema is a critical challenge in software development, particularly in the context of Model-Driven Development where the software being developed is the result of an evolution of models. For this reason, several techniques have been proposed to check the correctness of the behavioral conceptual schema. Most of these techniques are just concerned with identifying non-executable operations from the schema.

In contrast, we have proposed an approach both for identifying non-executable operations and also for providing the designer with information for fixing up the problem. This information is given in terms of missing structural events not initially stated in the operation postcondition, that ensure operation executability when taken into account. Thus, we extend previous approaches by providing the designer with relevant feedback when a non-executable operation is detected.

We have also implemented a prototype tool of our approach to analyze the scalability of our method in practice. We have shown that our tool is able to perform several complex tests in the conceptual schema of the DBLP case study in an average time of about 50 to 55 seconds.

This work can be extended in several directions. First, we would like to adapt our method to other modeling languages by taking advantage of the fact that our

underlying reasoning process is based on a logic formalization. Second, we would like to analyze the applicability of the dependencies proposed in this paper to other constraint-related problems such as integrity-constraint checking.

Acknowledgements This work has been partly supported by the Ministerio de Ciencia e Innovación under project TIN2011-24747 and by the FI grant from the Secreteria d'Universitats i Recerca of the Generalitat de Catalunya.

References

1. Olivé, A.: Conceptual Modeling of Information Systems. Springer, Berlin (2007)
2. Object Management Group (OMG): Unified Modeling Language (UML) Superstructure Specification, version 2.4.1. (2011) http://www.omg.org/spec/UML/.
3. Cabot, J., Clarisó, R., Riera, D.: Verifying UML/OCL operation contracts. In: Integrated Formal Methods, Springer (2009) 40–55
4. Object Management Group (OMG): Object Constraint Language (UML), version 2.3.1. (2012) http://www.omg.org/spec/OCL/.
5. Hamann, L., Hofrichter, O., Gogolla, M.: On integrating structure and behavior modeling with OCL. In: International Conference on Model Driven Engineering Languages & Systems (MODELS 2012), Springer (2012) 235–251
6. Soeken, M., Wille, R., Drechsler, R.: Verifying dynamic aspects of UML models. In: Design, Automation Test in Europe Conference Exhibition (DATE). (2011) 1–6
7. Roldán, M., Durán, F.: Dynamic validation of OCL constraints with mOdCL. In: International Workshop on OCL and Textual Modelling. (2011)
8. Queralt, A., Teniente, E.: Reasoning on UML conceptual schemas with operations. In: 21st International Conference on Advanced Information Systems Engineering (CAiSE'09). Volume 5565., Springer (2009) 47–62
9. Brucker, A.D., Wolff, B. HOL-OCL: a formal proof environment for UML/OCL. In: Fundamental Approaches to Software Engineering. Springer (2008) 97–100
10. Olivé, A.: Conceptual schema-centric development: A grand challenge for information systems research. In: Advanced Information Systems Engineering. Volume 3520 of Lecture Notes in Computer Science. Springer (2005) 1–15
11. Queralt, A., Teniente, E.: Verification and validation of UML conceptual schemas with OCL constraints. ACM TOSEM **21**(2) (2012) 13
12. Costal, D., Gómez, C., Queralt, A., Raventós, R., Teniente, E.: Improving the definition of general constraints in UML. Software & Systems Modeling **7**(4) (2008) 469–486
13. Olivé, A.: Integrity constraints checking in deductive databases. In: Proceedings of the 17th Int. Conference on Very Large Data Bases (VLDB). (1991) 513–523
14. Farré, C., Teniente, E., Urpí, T.: Checking query containment with the CQC method. Data & Knowledge Engineering **53**(2) (2005) 163–223
15. Planas, E., Olivé, A.: The DBLP case study (2006) http://guifre.lsi.upc.edu//DBLP.pdf.
16. Krieger, M.P., Knapp, A., Wolff, B.: Automatic and efficient simulation of operation contracts. In: 9th International Conference on Generative Programming and Component Engineering. GPCE '10, New York, USA, ACM (2010) 53–62
17. Albert, M., Cabot, J., Gómez, C., Pelechano, V.: Generating operation specifications from UML class diagrams: A model transformation approach. Data & Knowledge Engineering **70**(4) (2011) 365 – 389