Computing Repairs for Constraint Violations in UML/OCL Conceptual Schemas

Xavier Oriol[a,*], Ernest Teniente[a], Albert Tort[b]

[a] *Universitat Politècnica de Catalunya – Barcelona, Spain*
[b] *Sogeti España – Barcelona, Spain*

Abstract

Updating the contents of an information base may violate some of the constraints defined over the schema. The classical way to deal with this problem has been to reject the requested update when its application would lead to some constraint violation. We follow here an alternative approach aimed at automatically computing the repairs of an update, i.e., the minimum additional changes that, when applied together with the requested update, bring the information base to a new state where all constraints are satisfied. Our approach is independent of the language used to define the schema and the constraints, since it is based on a logic formalization of both, although we apply it to UML and OCL because they are widely used in the conceptual modeling community.

Our method can be used for maintaining the consistency of an information base after the application of some update, and also for dealing with the problem of fixing up non-executable operations. The fragment of OCL that we use to define the constraints has the same expressiveness as relational algebra and we also identify a subset of it which provides some nice properties in the repair-computation process. Experiments are conducted to analyze the efficiency of our approach.

Keywords: conceptual schema, UML/OCL, integrity constraint, repair

1. Introduction

An information system maintains a representation of the state of a real world domain in its information base (IB), where the domain is defined by means of a conceptual schema. A conceptual schema consists of two parts: a structural schema and a behavioral schema. The structural schema defines the structure of the IB and some integrity constraints, that is, some conditions that every IB should satisfy according to the domain [1]. The behavioral schema defines the events that might be applied to the IB to change its contents.

When a set of events is applied to an IB, it may be the case that some of the constraints in the structural schema are violated. To avoid so, several techniques have been proposed to efficiently check constraint satisfaction [2, 3, 4, 5, 6]. Using these techniques, it is possible to detect those violations committed by the events, but none of them is able to propose additional changes, i.e., *repairs*, that would bring the IB back to a consistent state where all constraints are satisfied. Therefore, according to those techniques, if after applying the events some constraint violation is detected, then, the changes implied by the events should necessarily be rolled-back to keep the IB consistent.

However, rolling back the update when there is some violation might not be always satisfactory since it might cause a mismatch between the IB and the reality it is intended to represent. Instead, we propose to compute the additional changes required to maintain the IB in a consistent state so that we are able to perform the update as required by the domain.

*Corresponding author
Email addresses: xoriol@essi.upc.edu (Xavier Oriol), teniente@essi.upc.edu (Ernest Teniente), albert.tort-pugibet@sogeti.com (Albert Tort)

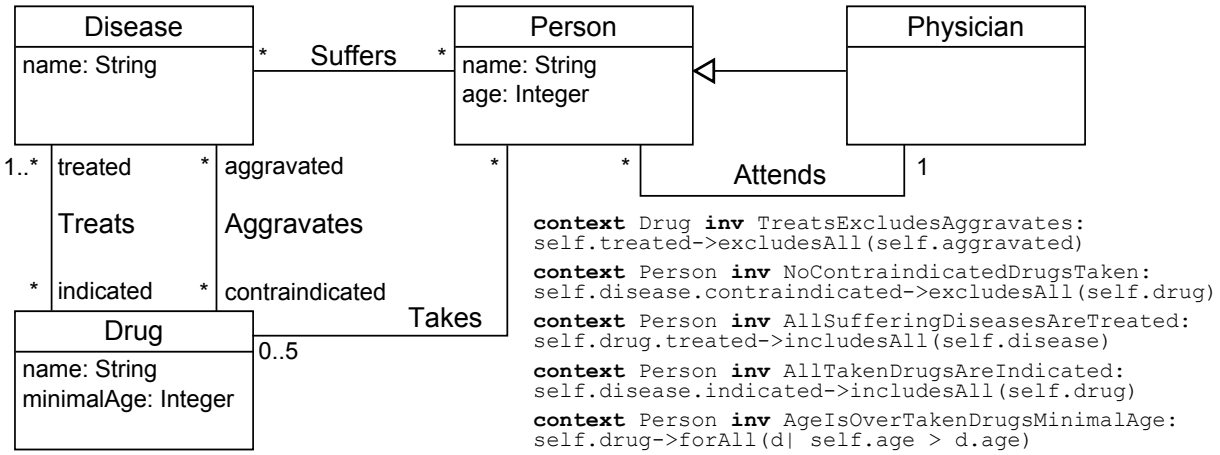

Figure 1: A UML/OCL schema for the domain of medical drugs taken by people

This leads to the main goal of this paper: given a structural schema with its constraints, an information base *IB*, and an update consisting of a set of events E to be applied to *IB*, we want to compute a repair R (i.e., an additional set of events) such that the application of $E \cup R$ to *IB* does not violate any of the constraints in the schema. Concretely, we are interested in computing *minimal* repairs, that is, repairs whose proper subsets are not repairs themselves.

In general, several repairs R_i may exist and the user should select the one to apply. The case in which there is only one empty repair $R_i = \emptyset$ corresponds the case in which E does not violate any constraint. In this manner, we can perform integrity checking along the same lines as the previous proposals by checking whether the returned set of repairs R_i is composed only of the empty set. It may also be the case that there is no repair R_i, meaning that E cannot be applied without necessarily violating the consistency of the IB.

Our approach can be applied independently of the language used for defining the structural conceptual schema and the integrity constraints since it is fully specified in terms of a logic formalization which can be drawn from different languages. However, for the purpose of this work, we assume that the structural schema is represented by means of a UML class diagram [7, 8] and that the constraints are specified as OCL constraints [9].

Computing repairs has attracted attention in Description Logics (DL) since the moment that *knowledge bases* have also been considered for providing support in the maintenance and evolution phase of information systems, thus requiring update operations to be performed directly on the knowledge base [10, 11]. However, these methods work with the *open world assumption* instead of the *closed world assumption* [12]. This is a crucial difference since an IB might be consistent under the open world assumption but inconsistent in the closed world, making these methods inapplicable in the latter.

In the context of UML, some work has been done to support change propagation in UML models [13, 14] and to resolve design model inconsistencies of UML schemas [15, 16, 17, 18]. These problems are specifically focused on fixing inconsistencies among the different UML models used for specifying an information system, but not the schema and the IB. To our knowledge, the unique work intended to do so is the OCLexec tool [19, 20], but without dealing the case of class instance deletion neither class instance *generalization*.

1.1. Motivating example

Consider the UML class diagram and OCL constraints in Figure 1. This schema specifies an information system about the medical drugs taken by people. In particular, for each *person*, the system stores the *physician* who is *attending* him/her. A person may *suffer* some *disease*, which can be *treated* or *aggravated* by means of some *drugs*. Each drug treats at least one disease and it also has a *minimal age* indication. A person takes at most five drugs.

OCL constraints provide the class diagram with additional semantics. *Treats-ExcludesAggravates* states that no drug is both indicated and contraindicated for the same disease. *NoContraindicatedDrugsTaken* asserts that nobody takes a drug that aggravates a disease he/she is suffering. *AllSufferingDiseasesAreTreated* ensures that everybody takes some drug indicated for each of his/her diseases, while *AllTakenDrugsAreIndicated* ensures that people do not take drugs that do not treat their diseases. *AgeIsOverTakenDrugsMinimalAge* guarantees that nobody takes some drug whose minimal indicated age is greater than their own. Additionally, for the sake of simplicity, we assume that the *name* attribute is unique in the classes person, disease and drug (i.e., there are no two instances of person, disease or drug, with the same name), and thus, we use it as their identifier.

Consider now the following IB for the class diagram in Figure 1:

$$\begin{array}{lll} treats(aspirin, headache) & disease(headache) & drug(aspirin) \\ treats(ibuprofen, headache) & disease(cold) & drug(ibuprofen) \\ treats(ibuprofen, cold) & person(john) & drug(loratadine) \\ treats(loratadine, cold) & & \end{array}$$

and suppose that new medical research has found that $ibuprofen$ is now contraindicated for *cold*. Then, such new fact should be added to the IB (or, more formally, the event $insert(aggravates(ibuprofen, cold))$ should be applied). The application of this event would, however, violate *TreatsExcludesAggravates*. In this case, the constraint violation can be repaired by additionally applying the event $delete(treats(ibuprofen, cold))$.

Things would become more complex if the requested event was to insert that *john* suffers a *cold*, which would violate *AllSufferingDiseasesAreTreated*. Here three possible repairs may be found: inserting that $john$ is taking $ibuprofen$; inserting that he is taking $loratadine$; or considering a new *drug* to be taken by $john$ which would also be indicated for a *cold*. It clearly follows from this example that trying to find repairs by hand would be unfeasible because of the number of possibilities to take into account. Moreover, it is not possible to say, a priori, whether there may be many, one or no repairs.

1.2. Contribution

To our knowledge, ours is the first approach able to automatically compute repairs when a set of insertion/deletion events is applied to an IB of a UML/OCL conceptual schema. In this way, we contribute to the conceptual-schema centric development grand challenge of enforcing integrity constraints in conceptual schemas [21] aimed at providing, among others, the automatic execution of the schema. This target is also aligned with the *Conceptual Modeling Programming Manifesto* [22], which argues that conceptual models should be executable.

Moreover, the kind of feedback we automatically provide to the user allows him/her to identify the additional changes needed to keep the IB consistent. Computing repairs is done without the user intervention, although he/she might be required to choose the best repair if several options exist.

In [23] we presented a particular application of this approach aimed at fixing up non-executable operations. We fix up a non-executable operation by means of computing the missing effects in the underspecified operation postcondition in a manner similar to how we compute the repairs for constraint violations. Computing the missing effects in underspecified operation postconditions corresponds to apply an extended interpretation of an operation [24], in which the operation is assumed to apply all the events specified in the postcondition and any additional event required to maintain the consistency of the IB. We provide now several additional contributions with regards to our previous work in [23]. The most important ones are the following:

- *Terminating fragment of OCL identified.* We prove that the fragment of OCL we dealt in our previous work, called $\text{OCL}_{\mathsf{UNIV}}$, ensures termination of the process for computing the repairs. We also analyze the efficiency of this fragment through experiments and we compare its expressiveness with that of OCL-Lite [25], a well-known decidable fragment of OCL.

- *Repairable fragment of OCL extended.* We extend the OCL fragment for which we can compute repairs to OCL_{FO}, i.e., a fragment of OCL which is expressively equivalent to relational algebra [26]. Due to the new

constraint expressiveness, we extend our previous algorithms to deal with this new complexity, and perform some experiments to show its efficiency.

- *Limitation of the solution space proposed.* In general, the number of expected solutions is exponential due to a combinatorial explosion intrinsic to the problem. However, we identify several strategies for keeping the number of solutions as low as possible by means of bounding techniques that removes solutions that are not acceptable in the domain.

2. Basic concepts and notation

On the following, we summarize the logic and conceptual modeling background used in this paper. Most of the logic concepts have been taken from[27], whereas the conceptual modeling definitions are based on [28].

Terms, atoms and literals. A *term* t is either a variable or a constant. An *atom* is formed by a n-ary *predicate* p together with n terms, i.e., $p(t_1, ..., t_n)$. We may write $p(\bar{t})$ for short. If all the terms $\bar{t}$ of an atom are constants, we say that the atom is *ground*. A literal l is either an atom $p(\bar{t})$, a negated atom $\neg p(\bar{t})$, or a built-in literal $t_i\ \omega\ t_j$, where ω is an arithmetic comparison (i.e., $<$,$\leq$,$=$,$\neq$).

Derived/base predicates. A predicate p is said to be *derived* if the boolean evaluation of an atom $p(\bar{t})$ depends on one or more derivation rules; otherwise it is said to be *base*. A *derivation rule* is a formula of the form:

$$\forall \bar{t}.\ p(\overline{t_p}) \leftarrow \phi(\bar{t})$$

where $\overline{t_p} \subseteq \bar{t}$. $p(\overline{t_p})$ is an atom called the *head* of the rule and $\phi(\bar{t})$ is a conjunction of literals called the *body*. We assume all derivation rules to be safe (i.e., all the variables appearing in the head or in a negated or built-in literal of the body also appear in a positive literal of the body) and non-recursive. Given several derivation rules with predicate p in the head, $p(\bar{t})$ is evaluated to true if and only if at least one of the bodies of those rules is true. We also say that an atom is base if its predicate is base too; otherwise it is derived.

Logic formalization of the UML schema. We formalize each class C in a class diagram with attributes $\{A_1, \ldots, A_n\}$ by means of a base atom $c(Oid)$ together with n atoms of the form $cA_i(Oid, A_i)$, each association R between classes $\{C_1, \ldots, C_k\}$ by means of a base atom $r(C_1, \ldots, C_k)$, and each association class R between classes $\{C_1, \ldots, C_k\}$ with attributes $\{A_1, \ldots, A_n\}$ by means of a base atom $r(Oid, C_1, \ldots, C_k)$ together with n atoms $rA_i(Oid, A_i)$, as proposed in [29].

Information base. An information system maintains a representation of the state of a domain in its *information base* (IB). An information base is a set of instances of the classes and associations defined in a conceptual schema. We represent the IB by means of a set of ground base atoms.

Constraint. A constraint is a logic assertion posed over a schema S that must be satisfied by the IB. Given a constraint c defined over a schema S and an *IB* of S, we say that *IB satisfies* c, i.e., $IB \models c$, if and only if c evaluates to true in *IB*. Otherwise, we say that *IB* violates c ($IB \not\models c$). We naturally extend this notion for a set of constraints C, i.e., $IB \models C$ iff $\forall c \in C.\ IB \models c$. Additionally, we say that *IB* of S is *consistent* when *IB* satisfies all the constraints defined in S; otherwise we say that the *IB* is *inconsistent*.

Structural events. A *structural event* is an elementary change in the population of a class or association (that is, a change of the IB). We consider six types of structural events: class instance insertion and deletion, association instance insertion and deletion, attribute instance insertion and deletion. We denote insertions by ι and deletions by δ.

Given a base atom $p(\overline{x})$, insertion structural events are formally defined by the formula $\forall \overline{x}(\iota p(\overline{x}) \leftrightarrow p^n(\overline{x}) \wedge \neg p(\overline{x}))$ and 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 information base, that is, the one obtained after applying the change.

Given a set of structural events E and an *IB*, the function $apply : \mathcal{E} \times \mathcal{IB} \rightarrow \mathcal{IB}$ returns the new IB^n resulting from applying E to *IB*.

Repair. Given an *IB*, a constraint c, and a set of structural events E such that $apply(E, IB) \not\models c$, we define a *repair* as another set of structural events R such that $apply(E \cup R, IB) \models c$. Given a set of constraints C,

R is a repair for IB, E, and C if and only if $apply(E \cup R, IB) \models C$. A repair R is *minimal* when no proper subset R' of R is also a repair, i.e., $\forall R' \subset R.\ apply(E \cup R', IB) \not\models C$.

Dependencies. A *Tuple-Generating Dependency (TGD)* is a formula of the form $\forall\, \overline{x}, \overline{z}.\ \varphi(\overline{x}, \overline{z}) \rightarrow \exists\, \overline{y}.\ \psi(\overline{x}, \overline{y})$. For our purposes, $\varphi(\overline{x}, \overline{z})$ will be a conjunction of literals and $\psi(\overline{x}, \overline{y})$ will be a conjunction of positive base atoms and optionally some built-in literals constraining its terms. 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 variation of TGDs where disjunctions are admitted in the conclusion of the rule. In particular, they follow the form: $\forall \overline{x}, \overline{z}.\ \phi(\overline{x}, \overline{z}) \rightarrow \bigvee \exists \overline{y}.\ \psi(\overline{x}, \overline{y})$. From now on, we omit the logic quantifiers since they can be understood by context.

A *Repair-Generating Dependency (RGD)* is a DED where the premise contains necessarily at least one structural event, whereas the conclusion is either a single structural event or a disjunction of several structural events, i.e., it has the form $ev_1(\overline{x_1}, \overline{y_1}) \vee \ldots \vee ev_k(\overline{x_k}, \overline{y_k})$, where each $ev_i(\overline{x_i}, \overline{y_i})$ is a structural event. An *Extended Repair-Generating Dependency* (eRGD) is an RGD where the conclusion is a disjunction of conjuncted structural events. That is, an eRGD has the form $E_1(\overline{x_1}, \overline{y_1}) \vee \ldots \vee E_k(\overline{x_k}, \overline{y_k})$, where each $E_i(\overline{x_i}, \overline{y_i})$ is a conjunction of structural events. An *Event-Dependency Constraint (EDC)* is an RGD in which the conclusion only contains the atom $\bot$.

3. Our approach in a nutshell

Given a structural schema S, a set of constraints C defined over S, an information base IB for S, and a set of structural events E to apply to IB, our goal is to automatically compute all minimal repairs R_i, that is, all those structural events that when applied with E to IB leads IB to a new consistent state.

Repairs are drawn from a set of rules, called *repair-generating dependencies* (RGDs for short), which are automatically obtained from each constraint in C. An RGD indicates which structural events will violate a constraint and the repairs that must be performed to remove that violation. In this way the set of RGDs captures all possible ways to detect and repair the integrity constraints of the schema. RGDs can be computed at compile time since they depend only on the definition of integrity constraints.

As an example, the RGDs obtained from the OCL constraint *TreatsExcludes-Aggravates* are the following:

$$aggravates(MD, D) \wedge \imath treats(MD, D) \rightarrow \delta aggravates(MD, D) \quad (1)$$

$$treats(MD, D) \wedge \imath aggravates(MD, D) \rightarrow \delta treats(MD, D) \quad (2)$$

$$\imath aggravates(MD, D) \wedge \imath treats(MD, D) \rightarrow \bot \quad (3)$$

The first rule states that the constraint will be violated if a medical drug MD is contraindicated to a disease D and we insert the fact that MD also treats D. The only way to repair such violation is by deleting the fact that MD is contraindicated to D. The semantics of the second rule is very similar to the first one. The third rule states that there is a violation without any possible repair when we insert the facts that MD both aggravates and treats D.

Given the set of RGDs drawn from the integrity constraints, we compute the different repairs R_i by means of chasing the RGDs with the information base IB and the set of events to apply E. Finally, the user is responsible for selecting the repair R_i to apply from the set of obtained repairs. The two-step process followed by our method is summarized in Figure 2. In the left-hand side of the figure, we see how RGDs are obtained from the OCL constraints specified by the domain expert. This is done through three intermediate stages (not shown in the figure): translating the schema and the constraints into its logic formalization, moving from this formalization to event-dependency constraints showing how constraints may be violated by means of events, and finally getting the RGDs from them.

The right-hand side illustrates the application of the chase to compute the different repairs. Chasing the RGDs is a dynamic step since it is performed at runtime each time a user wants to apply a set of events to the IB. Both RGD generation and chasing are fully automatic in the sense that no human intervention is required to perform them, although some expert user might need to choose the best repair in case the chase finds several options.

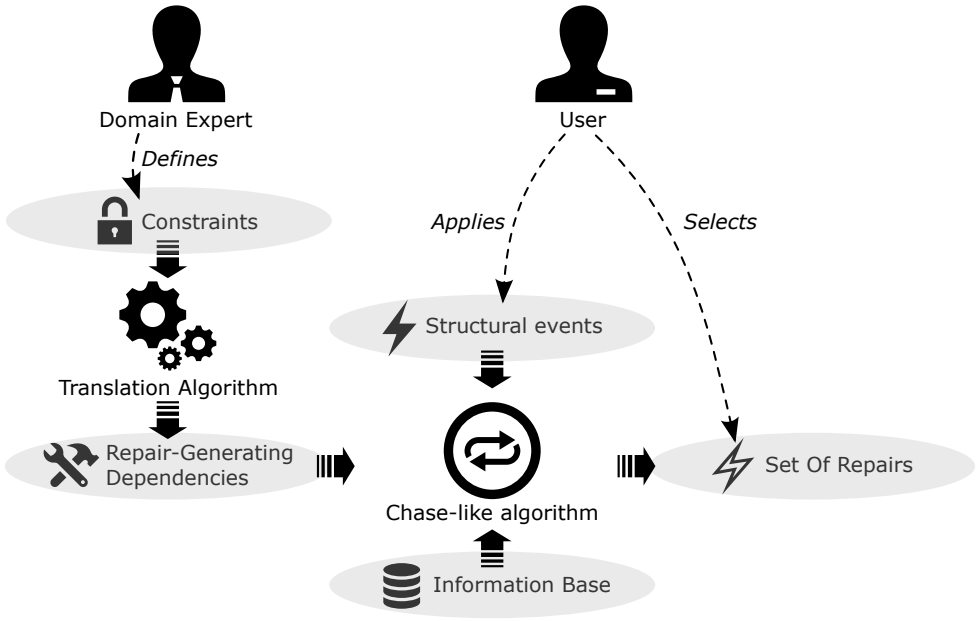

Figure 2: Overview of our approach

3.1. Specifying repairs in RGDs

We assumed in our previous work [23] that the conclusion of an RGD was a disjunction of single structural events, where each structural event stands for a different way to repair a constraint. Given the restricted fragment of OCL considered in that paper, this assumption was enough to capture all possible repairs since each OCL constraint defined in that fragment could be repaired through the application of a single event.

As an example, consider the minimum cardinality constraint stating that each drug must be indicated for at least one disease. If we delete that some drug MD is indicated for a disease D, where D is the only disease treated by MD, then we may repair the constraint violation by making MD indicated for a new disease D' or by deleting MD, but no additional events are required to repair the violation in any of the cases. These repairs may be specified by an RGD with the formula $\iota treats(MD, D') \vee \delta drug(D)$ in its conclusion.

Assuming that each constraint can be repaired through the application of a single event no longer holds when considering the OCL_{FO} subset of OCL [26], as we do in this paper. For instance, *AllSufferingDiseasesAreTreated* is violated when some person P stops taking the last medical drug MD treating some disease D that he is suffering. This violation may be repaired by applying two new events together: inserting the fact that P takes a new medical drug MD_2, and inserting the fact that MD_2 treats D. Therefore, the RGD's conclusion capturing this situation should be a disjunction of conjunctions of events (such as $\iota takes(P, MD_2) \wedge \iota treats(MD_2, D)$ in our example), where each conjunction of structural events also stands for a different repair of the constraint.

We capture these ideas by means of *extended repair-generating dependencies* (eRGDs), which is an extension of RGDs having disjunctions in the dependency conclusion.

3.2. Applying our approach to fix up non-executable operations

The approach we propose in this paper is also applicable to identify whether an operation is executable in a particular IB and to provide information about how to fix the problem when it is not. We addressed this problem in [23], for a certain fragment of OCL, where the information to fix up the problem was given in terms of the missing effects underspecified on the operation postcondition that would ensure that all constraints would be satisfied after the execution of the operation. Missing effects were given as a set of structural events R_i that should be applied together with those events E already implied by the postcondition. Such additional structural events R_i were computed by means of RGDs since they are *repairs*.

In this paper, besides dealing with a much more expressive fragment of OCL (i.e., OCL_{FO}) which increases the complexity of the problem, we have also relaxed the assumptions of [23] regarding the way the IB and the events used as input of our approach were obtained. This is shown in Figure 2, where we consider that the IB contains the current data of the application while the events to apply are given by the user.

However, our new approach can also be applied to fix up non-executable operations. Given a UML/OCL schema and an operation *Op* to be analyzed, we should first generate a consistent IB satisfying the operation precondition to test whether *Op* is executable in such IB. This IB can be automatically obtained by applying [29], manually given by the designer, or through a combination of both. The structural events from which to reason are drawn from the postcondition of *Op* according to the mapping defined in [30] and summarized in [23].

As an example, consider the following operation aimed at inserting the fact that a drug d aggravates a disease md:

Operation: addContraindication(d: Drug, md:Disease)
pre: -
post: md.contraindicated->includes(d)

Applying [29] from the parameters and the precondition of *addContraindication*, we get an information base $IB=\{drug(d_1), disease(md_1), treats(d_1, md_1)\}$; while with [30] we draw the set of structural events $E = \{\iota aggravates(d_1, md_1)\}$ from its postcondition. Now, we can apply our approach exactly as stated in Figure 2. In this scenario, the chase would return a repair $R_i = \{\delta treats(d_1, md_1)\}$ because of RGD 2. Thus, *addContraindication* is not executable in *IB* since it would necessarily violate the constraint *TreatsExcludesAggravates*. To fix-up such operation, we should modify the postcondition of the operation to additionally imply the event $\delta treats(d_1, md_1)$.

Besides fixing up non-executable operation contracts, this particular application of our approach permits computing at run time the *extended interpretation* of an operation [24]. An extended interpretation of an operation contract assumes a reactive behavior of the operation when its application induces some constraint violation. In particular, it assumes that an operation execution additionally applies any set of structural events required to keep the IB consistent if some constraint is violated, although these events might not be specified in the postcondition.

4. Obtaining RGDs and *e*RGDs

As we have seen in the previous section, repair-generating dependencies (extended or not) are a crucial component in our approach since they capture the different ways a constraint can be repaired. Given an OCL constraint in the structural conceptual schema, its associated dependencies are automatically computed according to the following steps:

1. Encoding the OCL constraint as a *logic denial*.
2. Obtaining the *event dependency constraints* (EDCs) from the denial.
3. Drawing the (extended) repair-generating dependencies from the EDCs.

The translation from OCL constraints to (extended) RGDs is static in the sense that it is performed only once at compilation time, when the schema and the OCL constraints have been defined.

At the end of the translation, we obtain a set of repair-generating dependencies (RGDs) or a set of extended repair-generating dependencies (*e*RGDs) depending on the expressiveness of the fragment of OCL considered. We have identified two fragments of OCL for this purpose: $\text{OCL}_{\mathsf{UNIV}}$ and OCL_{FO}.

OCL_{UNIV}. The main feature of the $\text{OCL}_{\mathsf{UNIV}}$ constraints is that they give rise to a particular case of RGDs, i.e., dependencies with a single event in each disjunct of the conclusion, where all variables appearing in the conclusion are universally quantified. $\text{OCL}_{\mathsf{UNIV}}$ is defined by the syntax:

ExpBool	::=	*ExpBool* `and` *ExpBool*	| *ExpBool* `or` *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*`->isUnique(`*Attrib*`)`	| *Class*`.allInstances()->exists(`*IdEq*`)`
Path	::=	*Var*.*Navigation*	| *Class*`.allInstances().`*Navigation*
Navigation	::=	*Role*.*Navigation*	| `oclAsType(`*Class*`).`*Navigation*
		| *Role*	| *Attrib*
		| `oclAsType(`*Class*`)`	

where *OpComp* is an arithmetic comparison (i.e., $<$,$\leq$,$=$,$\neq$), *Constant* is an OCL literal (e.g. integers such as `1`, strings such as `"Hello world!"`, etc.), *Var* is an iteration variable appearing inside a `forAll` or the special OCL variable `self`. In `exists`, *IdEq* is an equality comparison between the properties identifying the source *Class* (in particular, OIDs for classes, association ends for association classes) and some OCL variables (e.g. `self` or iterator variables from `forAll`). This limited `exists` operation is included to simulate an `includes`/`includesAll` with (reified) *n-ary* associations where $n > 2$. As expected, *Class*, *Role* and *Attrib* are names of classes, association roles and class attributes appearing in the UML class diagram.

OCL_{UNIV} is the fragment of OCL that we considered in our previous work [23], where we used RGDs for fixing-up non-executable operations.

OCL_{FO}. OCL_{FO} is a fragment of OCL expressively equivalent to relational algebra [26] that subsumes OCL_{UNIV}. In particular, it is the fragment of OCL limited to first-order constructs, that is, OCL excluding general aggregation such as `sum`, the transitive closure `closure` and where any collection is a *set*. OCL constraints defined according to this fragment give rise to *e*RGDs, i.e., dependencies whose conclusion is a disjunction of conjunctions of structural events, and where variables in the conclusion may be existentially quantified.

Step 1 of the translation process is the same for OCL_{UNIV} and OCL_{FO} constraints. However, steps 2 and 3 depend on the OCL fragment considered. In the rest of this section we define in detail all these three steps.

4.1. Encoding the OCL constraints as logic denials

We first encode each OCL constraint as a logic denial as proposed in [29]. These logic denials are written over the predicates of the logic formalization of the UML schema that has been described in Section 2. In our example of Figure 1, the formalization for the UML class diagram is:

$$disease(D), drug(MD), drugAge(MD, A), person(P), personAge(P, A)$$
$$physician(P), suffer(P, D), treats(MD, D), aggravates(MD, D)$$
$$takes(P, MD), attends(P1, P2)$$

where we use, without loss of generality, the identifier attribute of a class as its *oid*. In this way, we can omit the predicate corresponding to that attribute.

Given the previous schema, OCL constraints *TreatsExcludesAggravates* and *AllSufferingDiseasesAreTreated* would be encoded into the following denials [1]:

$$treats(MD, D) \wedge aggravates(MD, D) \rightarrow \bot \quad (4)$$
$$suffers(P, D) \wedge \neg diseaseTreated(P, D) \rightarrow \bot \quad (5)$$
$$diseaseTreated(P, D) \leftarrow takes(P, MD) \wedge treats(MD, D) \quad (6)$$

[1] Note that if a denial contains a derived predicate, like in rule 5, then additional derivation rules are required to define them.

Denial 4 is obtained from *TreatsExcludesAggravates* and states that there might not be a medical drug *MD* that treats and aggravates the same disease *D*. Denial 5, drawn from *AllSufferingDiseasesAreTreated*, prevents a person *P* to suffer from a disease *D* without taking any medical drug *MD* treating *D*. This last denial makes use of a derived predicate *diseaseTreated*, which is defined in the derivation rule 6.

In addition to the $\text{OCL}_{\mathsf{UNIV}}$ and OCL_{FO} constraints, the encoding proposed in [29] also defines how to obtain denials for graphical/implicit constraints in the UML schema such as min/max cardinality constraints, referential integrity constraints from associations, hierarchies, and disjoint/complete constraints on hierarchies. Therefore, our approach is also able to compute repairs for violations of such kinds of constraints since we obtain repair-generating dependencies from the denials, regardless of whether they come from a constraint of the UML schema or an OCL constraint.

4.2. Obtaining RGDs from $\text{OCL}_{\mathsf{UNIV}}$ constraints

Logic denials obtained in the previous step have a different form depending on whether its original OCL constraint was written in $\text{OCL}_{\mathsf{UNIV}}$ or OCL_{FO}. As we are going to prove in Section 6, logic denials coming from $\text{OCL}_{\mathsf{UNIV}}$ constraints are guaranteed to contain only base predicates. This simplifies its translation to RGDs in comparison to those denials coming from OCL_{FO} constraints, which usually contain negative derived literals like in rule 5. We explain in this section how to obtain RGDs for $\text{OCL}_{\mathsf{UNIV}}$ constraints.

4.2.1. Obtaining EDCs for $\text{OCL}_{\mathsf{UNIV}}$ constraints

An event dependency constraint (EDC) identifies a particular situation in which the original $\text{OCL}_{\mathsf{UNIV}}$ constraint would be violated in an information base IB^n resulting from applying some set of structural events E to some initial information base IB. Therefore, each denial constraint obtained in the previous step will be translated into several EDCs, corresponding to the different ways the constraint may be violated depending on the applied events.

The main idea for obtaining the EDCs is to replace each literal in the denial constraint by the expression that evaluates it in the new IB^n. Positive and negative literals in the denial are handled in a different way, according to the following formulas:

$$\forall \overline{x}.\ p^n(\overline{x}) \leftrightarrow (\iota p(\overline{x})) \vee (\neg\delta p(\overline{x}) \wedge p(\overline{x})) \tag{7}$$

$$\forall \overline{x}.\ \neg p^n(\overline{x}) \leftrightarrow (\delta p(\overline{x})) \vee (\neg\iota p(\overline{x}) \wedge \neg p(\overline{x})) \tag{8}$$

Rule 7 states that an atom $p(\overline{x})$ (e.g. $person(john)$) will be true in the new IB^n if its insertion structural event has been applied (e.g. person *john* has been inserted) or if it was already true in the initial *IB* and its deletion structural event has not been applied (e.g. the person *john* existed in the initial *IB* and it has not been deleted). In an analogous way, rule 8 states that $p(\overline{x})$ will not hold in the new IB^n if it has been deleted or if it was already false and it has not been inserted.

By applying the substitutions above, we get a set of EDCs that state all possible ways to violate a constraint by means of the possible structural events of the schema. EDCs are grounded on the idea of *event rules* which were defined in [31] 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 *getEventDependencyConstraints* performing this transformation is shown in Algorithm 1.

Intuitively, the algorithm interprets each literal p as p^n and it performs a replacement according to the definition given by formulas 7 and 8. One of the EDCs generated corresponds to a dependency that would be activated just in case the constraint was violated in the initial *IB* since it has no positive structural event. Using the common assumption that the initial *IB* state is consistent, we can safely delete such EDC. This deletion is done in the *removeEDCWithout-PositiveEvents* function.

Applying Algorithm 1 to the constraint *TreatsExcludesAggravates*, encoded in the first step into the denial $treats(MD, D) \wedge aggravates(MD, D) \rightarrow \bot$, we get the following event-dependency constraints:

$$treats(MD, D) \wedge \neg\delta treats(MD, D) \wedge \iota aggravates(MD, D) \rightarrow \bot \tag{9}$$

$$\iota treats(MD, D) \wedge aggravates(MD, D) \wedge \neg\delta aggravates(MD, D) \rightarrow \bot \tag{10}$$

$$\iota treats(MD, D) \wedge \iota aggravates(MD, D) \rightarrow \bot \tag{11}$$

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

$EDC := \{\emptyset \rightarrow \bot\}$
for all Literal p in $premise$ **do**
  $EDC_{Pre} := EDC$
  $EDC := \emptyset$
  for all Dependency $premise_{Pre} \rightarrow \bot$ in EDC_{Pre} **do**
    if p is Built-in literal **then**
      $EDC := EDC \cup \{premise_{Pre} \wedge p \rightarrow \bot\}$
    else
      if p is positive **then**
        $EDC := EDC \cup \{premise_{Pre} \wedge p \wedge \neg\delta p \rightarrow \bot\} \cup \{premise_{Pre} \wedge \iota p \wedge \bot\}$
      else
        $EDC := EDC \cup \{premise_{Pre} \wedge \delta p \rightarrow \bot\} \cup \{premise_{Pre} \wedge \neg\iota p \wedge \neg p \rightarrow \bot\}$
      end if
    end if
  end for
end for
EDC.removeEDCWithoutPositiveEvents()
return EDC

Rule 9 is an EDC stating that the constraint will be violated for a medical drug *MD* and a disease *D* if we insert the fact that *MD* aggravates *D* but we do not delete at the same time the fact that *MD* treats *D*. EDC 10 identifies a violation to happen when we insert the fact that *MD* treats *D* without deleting the fact that *MD* aggravates *D*. EDC 11 states that *TreatExcludesAggravates* will be violated if we insert the facts that *MD* treats and aggravates *D* at the same time.

4.2.2. Drawing RGDs for OCL_{UNIV} constraints

EDCs points out 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 the repairs for that violated constraint. We transform EDCs into RGDs for this purpose by means of the algorithm *getRepairGeneratingDependencies*, 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, we obtain the RGDs by removing the negated structural events of the EDC and placing them positively in the conclusion. If there is more than one negated structural event in the EDC, then there is more than one possible repair. Thus, these events are all placed in the conclusion of the RGD as a disjunction. Note that we obtain exactly one RGD for each EDC.

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

$new\_Conclusion := \bot$
$new\_Premise := \top$
for all Literal p in $premise$ **do**
  if p is negative structural event **then**
    $new\_Conclusion := new\_Conclusion \vee positive(p)$
  else
    $new\_Premise := new\_Premise \wedge p$
  end if
end for
return $new\_Premise \rightarrow new\_Conclusion$

Applying Algorithm 2 to the EDCs defined by the rules 9, 10, 11, we get the following RGDs:

$$treats(MD, D) \wedge \iota aggravates(MD, D) \rightarrow \delta treats(MD, D) \quad (12)$$

$$\iota treats(MD, D) \wedge aggravates(MD, D) \rightarrow \delta aggravates(MD, D) \quad (13)$$

$$\iota treats(MD, D) \wedge \iota aggravates(MD, D) \rightarrow \bot \quad (14)$$

The first RGD states that whenever we insert a fact that a medical drug *MD* aggravates a disease *D* when the current information base states that *MD* treats *D*, then, we should delete the fact that *MD* treats *D*. Similarly, the second one establishes that when a medical drug *MD* aggravates some disease *D* and we insert the fact that *MD* treats *D*, then, we should delete the fact that *MD* aggravates *D*. The third RGD

is exactly its original EDC 11, meaning that there is no possible way to repair that situation. That is, the structural events $\iota treats(MD, D) \wedge \iota aggravates(MD, D)$ cannot happen together without necessarily violating a constraint.

4.3. Obtaining eRGDs from OCL$_{FO}$ constraints

Denial constraints obtained from OCL$_{\mathsf{FO}}$ constraints usually contain negative derived base predicates. Therefore, we have to extend the algorithms proposed in the previous section in order to obtain the repair-generating dependencies for such constraints. As we are going to see, in this case we will mostly obtain eRGDs rather than RGDs due to the negative derived predicates.

4.3.1. Obtaining EDCs for OCL$_{FO}$ constraints

If we apply Algorithm 1 strictly to the denials obtained from OCL$_{\mathsf{FO}}$ constraints, we would get EDCs which contain derived events (i.e., events of insertion/deletion on the population of a derived predicate). Therefore, in this case, besides translating denials into EDCs, we also need to define the derivation rules for computing the derived events (i.e., rules for computing which structural events cause an insertion/deletion on the population of some derived predicate).

As an example, consider again the denial 5 we got from the OCL$_{\mathsf{FO}}$ constraint *AllSufferingDiseasesAre-Treated*. Applying Algorithm 1 to it, we get the following rules:

$$suffers(P, D) \wedge \neg\delta suffers(P, D) \wedge \delta diseaseTreated(P, D) \rightarrow \bot \quad (15)$$

$$\iota suffers(P, D) \wedge \neg diseaseTreated(P, D) \wedge \neg\iota diseaseTreated(P, D) \rightarrow \bot \quad (16)$$

$$\iota suffers(P, D) \wedge \delta diseaseTreated(P, D) \rightarrow \bot \quad (17)$$

Notice that we now have two derived event predicates: $\iota diseaseTreated$ and $\delta diseaseTreated$. Then, we also need to define their derivation rules. Intuitively, $\iota diseaseTreated(P,D)$ will be true if $diseaseTreated(P,D)$ was false in the initial IB and $diseaseTreated(P,D)$ is true in the new IB^n; and similarly for $\delta diseaseTreated(P,D)$. Therefore, we also need to define the following derivation rules for computing such derived events:

$$\iota diseaseTreated(P, D) \leftarrow \neg diseaseTreated(P, D) \wedge diseaseTreated^n(P, D) \quad (18)$$

$$\delta diseaseTreated(P, D) \leftarrow diseaseTreated(P, D) \wedge \neg diseaseTreated^n(P, D) \quad (19)$$

Predicate $diseaseTreated$ is already well defined in 6 but $diseaseTreated^n$ is not yet defined. This predicate corresponds to evaluating $diseaseTreated$ in the new IB^n which can be done by means of evaluating the body of its original derivation rule in IB^n, in particular:

$$diseaseTreated^n(P, D) \leftarrow takes^n(P, MD) \wedge treats^n(MD, D) \quad (20)$$

Now we can unfold the predicates $takes^n$ and $treats^n$ with the rules defined in 7 and 8, obtaining the rules:

$$diseaseTreated^n(P, D) \leftarrow takes(P, MD) \wedge \neg\delta takes(P, MD) \wedge treats(MD, D) \wedge \neg\delta treats(MD, D) \quad (21)$$

$$diseaseTreated^n(P, D) \leftarrow takes(P, MD) \wedge \neg\delta takes(P, MD) \wedge \iota treats(MD, D) \quad (22)$$

$$diseaseTreated^n(P, D) \leftarrow \iota takes(P, MD) \wedge treats(MD, D) \wedge \neg\delta treats(MD, D) \quad (23)$$

$$diseaseTreated^n(P, D) \leftarrow \iota takes(P, MD) \wedge \iota treats(MD, D) \quad (24)$$

For optimization purposes, consider again the rule 19 that defines the derived event $\delta diseaseTreated$ after unfolding the predicate $diseaseTreated$:

$$\delta diseaseTreated(P, D) \leftarrow takes(P, MD) \wedge treats(MD, D) \wedge \neg diseaseTreated^n(P, D) \quad (25)$$

If *diseaseTreated(P, D)* evaluates to true in the initial *IB*, but it evaluates to false in the new IB^n, then, necessarily one of the positive atoms of its body has been removed in the new IB^n. Thus, we can replace this rule by the rules:

$$\delta diseaseTreated(P,D) \leftarrow \delta takes(P,MD) \wedge treats(MD,D) \wedge \neg diseaseTreated^n(P,D) \quad (26)$$

$$\delta diseaseTreated(P,D) \leftarrow takes(P,MD) \wedge \delta treats(MD,D) \wedge \neg diseaseTreated^n(P,D) \quad (27)$$

These two rules are logically equivalent to the rule 25 but they are more efficient to evaluate. This is because they are defined in terms of structural events rather than usual instances from the *IB*, and since we expect the number of structural events to apply to be much lower than the number of instances in the *IB*, evaluating the body of these two rules is faster than computing rule 25 .

Algorithm 3 applies all these transformations and optimizations. That is, it receives as input a derivation rule $head \leftarrow body$ and returns as output the set of necessary derivation rules to compute the $\iota head$ and $\delta head$ derived events.

Algorithm 3 getInsertionDeletionDerivedEventRules($head \leftarrow body$)

```
% Create the New State Derivation Rules
NewStateDerivRules := {head^n ← ∅}
for all Literal p in body do
    NewStateDerivRules_Pre := NewStateDerivRules
    NewStateDerivRules := ∅
    for all DerivationRule head^n_Pre ← body^n_Pre in NewStateDerivRules_Pre do
        if p is Built-in-literal then
            NewStateDerivRules := NewStateDerivRules ∪ {head^n_Pre ← body^n_Pre ∧ p}
        else
            if p is positive then
                NewStateDerivRules := NewStateDerivRules ∪ {head^n_Pre ← body^n_Pre ∧ p ∧ ¬δp} ∪ {head^n_Pre ← body^n_Pre ∧ ιp}
            else
                NewStateDerivRules := NewStateDerivRules ∪ {head^n_Pre ← body^n_Pre ∧ δp} ∪ {head^n_Pre ← body^n_Pre ∧ ¬ιp ∧ ¬p}
            end if
        end if
    end for
end for
% Create the Insertion Event Rules
InsertionDerivRules := ∅
for all DerivationRule head^n ← body^n in NewStateDerivRules do
    InsertionDerivRules := InsertionDerivRules ∪ {ιhead ← body^n ∧ ¬head}
end for
% Create the Deletion Event Rules
DeletionDerivRules := ∅
for all Literal p in body do
    DeletionDerivRules := DeletionDerivRules ∪ {δhead ← (body \ {p}) ∧ δp ∧ ¬head^n}
end for
return InsertionDerivRules ∪ DeletionDerivRules ∪ NewStateDerivRules
```

The algorithm starts by creating the new state derivation rules (e.g. the derivation rules of $diseaseTreated^n$) following exactly the same strategy as followed in Algorithm 1 to obtain the EDCs: interpreting each literal p in *body* as p^n and unfolding according to rules 7 and 8.

Then we create the insertion derivation rules (e.g. $\iota diseaseTreated$) by means of adding, for each derivation rule of the new state predicate, a new rule containing the same body but adding a literal to check that the *head* was not true in the previous state (e.g. $\neg diseaseTreated$).

Finally, we create the deletion derivation rules (e.g. $\delta diseaseTreated$) by means of iterating each literal of the original derivation rule *body*, replacing the literal with its deletion literal (e.g. *takes* for $\delta takes$ and *treats* for $\delta treats$) and adding a literal to check that the head does not exist in the new state (e.g. $\neg$diseaseTreated^n).

4.3.2. Drawing eRGDs for OCL$_{FO}$ constraints

EDCs coming from OCL$_{FO}$ constraints will in general contain negative derived events, like rule 16 which contains $\neg\iota diseaseTreated(P,D)$. Therefore, if we build repair-generating dependencies by just applying the criteria used in Section 4.2.2 of removing negative events from the premise and placing them positively into the conclusion, we would end up with derived events in the conclusion of the rule. Then, this rule would not be a repair-generating dependency since RGDs require all the literals in the conclusion to be base. This

is a necessary condition to ensure that chasing the RGDs computes repairs that contain only structural events and no derived events. Therefore, we require an additional transformation to remove derived events from the conclusion of such rules.

As an example, by applying Algorithm 2 to the previously stated EDC 16 we would obtain:

$$\iota suffers(P, D) \land \neg diseaseTreated(P, D) \rightarrow \iota diseaseTreated(P, D) \tag{28}$$

To ensure that all literals in the conclusion of this rule are base, we can unfold $\iota diseaseTreated(P, D)$ and obtain:

$$\begin{aligned}\iota suffers(P, D) \land \neg diseaseTreated(P, D) \rightarrow \\ (takes(P, MD) \land \neg\delta takes(P, MD) \land \iota treats(MD, D) \land \neg diseaseTreated(P, D)) \lor \\ (\iota takes(P, MD) \land treats(MD, D) \land \neg\delta treats(MD, D) \land \neg diseaseTreated(P, D)) \lor \\ (\iota takes(P, MD) \land \iota treats(MD, D) \land \neg diseaseTreated(P, D))\end{aligned}$$

Note that $\neg diseaseTreated(P, D)$ appears both in the premise and the conclusion of the previous rule. For optimization purposes, we can safely delete the literal from the conclusion since its truth evaluation is guaranteed by the premise.

$$\begin{aligned}\iota suffers(P, D) \land \neg diseaseTreated(P, D) \rightarrow \\ (takes(P, MD) \land \neg\delta takes(P, MD) \land \iota treats(MD, D)) \lor \\ (\iota takes(P, MD) \land treats(MD, D) \land \neg\delta treats(MD, D)) \lor \\ (\iota takes(P, MD) \land \iota treats(MD, D))\end{aligned}$$

Due to the unfolding, we may have added in the conclusion of the eRGD some literals that are not events (e.g. $takes(P, MD)$) and negative events (e.g. $\neg\delta takes(P, MD)$). Again, these literals are not allowed to appear in the eRGD conclusion and we also have to remove them. This can be done by using the following transformation defined in [32].

Given a disjunctive embedded dependency with a negated literal in its conclusion like:

$$\phi(\overline{x}) \rightarrow \psi_1(\overline{x_1}) \land \neg p(\overline{x_p}) \lor ... \lor \psi_n(\overline{x_n})$$

we can build two new disjunctive embedded dependencies with no negated literals in the conclusion that are equivalent to the initial one:

$$\begin{aligned}&\phi(\overline{x}) \rightarrow \psi_1(\overline{x_1}) \land forbidP(\overline{x_p}) \lor ... \lor \psi_n(\overline{x_n}) \\ &forbidP(\overline{x_p}) \land p(\overline{x_p}) \rightarrow \bot\end{aligned}$$

Intuitively, if we decide to repair the violation of ϕ by means of $\psi_1(\overline{x_1}) \land forbidP(\overline{x_p})$, then $p(\overline{x_p})$ must necessarily evaluate to false (otherwise, we would violate the second dependency defined). In this way, we simulate the behavior of the original dependency with a negated literal in the conclusion. For our purposes, we use $\iota forbidP$ instead of $forbidP$ in order to use the syntax of structural events.

In a similar way, we extend this transformation for a positive literal (not representing a structural event) in the conclusion. Given a dependency:

$$\phi(\overline{x}) \rightarrow \psi(\overline{x_1}) \land p(\overline{x_p}) \lor ... \lor \psi_n(\overline{x_n})$$

we can build two new disjunctive embedded dependencies taking out one of the positive literals of the conclusion:

$$\begin{aligned}&\phi(\overline{x}) \rightarrow \psi_1(\overline{x_1}) \land \iota containP(\overline{x_p}) \lor ... \lor \psi_n(\overline{x_n}) \\ &\iota containP(\overline{x_p}) \land \neg p(\overline{x_p}) \rightarrow \bot\end{aligned}$$

In our previous example, using these two transformations to remove undesired literals from the conclusion, we obtain the following final eRGD together with four new EDCs:

$$\begin{aligned}
&\iota suffers(P,D) \wedge \neg diseaseTreated(P,D) \rightarrow \\
&\qquad (\iota containTakes(P,MD) \wedge \iota forbidDelTakes(P,MD) \wedge \iota treats(MD,D)) \vee \\
&\qquad (\iota takes(P,MD) \wedge \iota containTreats(MD,D) \wedge \iota forbidDelTreats(MD,D)) \vee \\
&\qquad (\iota takes(P,MD) \wedge \iota treats(MD,D)) \\
&\iota containTakes(P,MD) \wedge \neg takes(P,MD) \rightarrow \bot \\
&\iota forbidDelTakes(P,MD) \wedge \delta takes(P,MD) \rightarrow \bot \\
&\iota containTreats(P,MD) \wedge \neg treats(P,MD) \rightarrow \bot \\
&\iota forbidDelTreats(P,MD) \wedge \delta treats(P,MD) \rightarrow \bot
\end{aligned}$$

In Algorithm 4, which is an extension of Algorithm 2, we formally define how to obtain the eRGDs of a given EDC by applying all these transformations. It receives as input an EDC and returns as output a set of eRGDs. The number of eRGDs obtained depends on the derived literals appearing in the EDC.

Algorithm 4 getExtendedRepairGeneratingDependencies($premise \rightarrow \bot$)

```
result := ∅
if premise contains a positive derived literal p then
    for all unfoldedPremise in unfoldingPremise(p, premise) do
        result := result ∪ getRepairGeneratingDependencies(unfoldedPremise → ⊥)
    end for
else
    new_Conclusion := ⊥
    new_Premise := ⊤
    % Place negative literals to the conclusion
    for all Literal p in premise do
        if p is negative event then
            new_Conclusion := new_Conclusion ∨ positive(p)
        else
            new_Premise := new_Premise ∧ p
        end if
    end for
    % Apply all the possible unfoldings
    for all Positive derived literal p in new_Conclusion do
        new_Conclusion := unfoldingConclusion(p, new_Conclusion)
    end for
    % Remove negative literals from the conclusion
    for all Negative literal p in new_Conclusion do
        new_Conclusion := replace(¬p, ιforbidP, new_Conclusion)
        result := result ∪ getExtendedRepairGeneratingDependencies(ιforbidP ∧ p → ⊥)
    end for
    % Remove non structural event literals from the conclusion
    for all Non structural event literal p in new_Conclusion do
        new_Conclusion := replace(p, ιcontainP, new_Conclusion)
        result := result ∪ {ιcontainP ∧ ¬p → ⊥}
    end for
    result := result ∪ {new_Premise → new_Conclusion}
end if
return result
```

The algorithm starts by checking the existence of positive derived literals in the premise. If this is the case, the algorithm unfolds the literal and recursively calls the algorithm until all the literals are base. Since, in our settings, the predicates are non-recursive, this recursion is guaranteed to terminate.

Then, the algorithm performs four loops, each one corresponding to a different transformation.

The first loop moves the negative event literals (structural or derived) from the premise to the conclusion. This is exactly the transformation defined in Algorithm 2 but extended to derived events.

The second loop applies the usual unfolding for derived literals placed in the conclusion, which is also guaranteed to terminate.

The third loop removes negative literals from the conclusion using the transformation in [32]. This transformation encompasses a recursive call to the algorithm for some predicate p. This recursion directly

terminates if the predicate p is base since the formula $\iota forbidP \wedge p \rightarrow \bot$ would need no more transformations. Again, the absence of recursive predicates guarantees that at some point predicate p will be base and, thus, the recursion will be finite.

The last loop removes non-structural event literals from the conclusion. We *replace* any non-structural event literal p for $\iota containP$ and add a new EDC $\iota containP \wedge \neg p \rightarrow \bot$. In this case, we do not need to call the algorithm recursively to translate this EDC to eRGDs because we know that, since p is base, no transformation would be applied to the rule.

Since all the recursions and loops are guaranteed to terminate, the algorithm terminates.

Moreover, the eRGDs generated by the algorithm are equivalent to the input EDCs in the following sense: for any information base IB and set of structural events E, $apply(E, IB)$ satisfies the EDCs if and only if $apply(E', IB)$ also satisfies all the generated eRGDs, where E' is the set of structural events E with possibly some extra atoms of the form $\iota containP$, $\iota forbidP$. We now prove that this kind of logical equivalence is preserved along all the transformations we apply.

Clearly, unfolding a formula is known to be logically equivalent to its original one. The transformation for moving negated literals from the premise to the conclusion is based on the logical equivalence:

$$\phi \wedge \neg p \rightarrow \psi \equiv \neg\phi \vee p \vee \psi \;\; \equiv \phi \rightarrow p \vee \psi$$

The transformation consisting in the replacement of $\neg p$ for $\iota forbidP$ and p for $\iota containP$ is based on forcing $forbidP \equiv \neg p$ and $containP \equiv p$. On one hand, the additionally created EDCs ensure that $\iota forbidP \implies \neg p$ and $\iota containP \implies p$. On the other hand, we are able to add as many ground atoms of $\iota containP$ and $\iota forbidP$ as we require in order to force $\neg p \implies \iota forbidP$ and $p \implies \iota containP$ (since adding them does not cause the violation of any eRGD besides $\iota containP \wedge \neg p \rightarrow \bot$ and $\iota forbidP \wedge p \rightarrow \bot$ since these literals do not appear in any other RGD premise). Thus, we achieve to force $p \equiv containP$ and $\neg p \equiv forbidP$. Note that we only need to create a finite number of instances of $forbidP$ because the rules are safe.

4.4. Removing/tuning e*RGDs to limit the number of obtained repairs*

The disjunctions and the existential variables in the conclusion of eRGDs cause the number of repairs to grow exponentially since all of them specify alternative ways to repair a constraint. Therefore, if two constraints are violated and each one may be repaired by means of m and n different possibilities respectively, we may expect about $m*n$ repairs in general. Therefore, it becomes necessary to take additional information into account for keeping the number of repairs as low as possible. This can be done by considering information from the domain directly stated in the UML schema or directly provided by domain experts to remove some eRGDs or some of its disjuncts.

addOnly and frozen stereotypes. A *changeability* stereotype can be specified for attributes and roles of the schema. Its default value is *changeable*, stating that the attribute or role value can be updated whenever desired, but other possible values are *addOnly*, stating that they can never be removed, and *frozen*, stating that they cannot be changed after the object is initialized. In our example, the class *Disease* can be defined to be add-only. [2]

Invalid repairs. Additionally, a user may define some kind of structural events to be *invalid repairs* for some concrete structural events to be applied. An invalid repair is an event type that cannot appear in any repair R_i. For example, given the structural event $\iota suffers(john, cold)$, it might be invalid to repair any violation caused by this event by means of adding new medical drugs in the system (in this manner, we limit the repairs to using medical drugs that currently exist in our system).

Both invalid repairs and *addOnly* and *frozen* stereotypes invalidate certain events in the domain and thus, they should not be considered in the repair-generating dependencies (either RGDs or eRGDs). Intuitively, if such an event appears in the premise of the dependency, then the dependency can be removed because

[2] In UML 1.x the *changeability* property was part of the UML metamodel. However, since this property has disappeared in UML 2.x, it now needs to be defined as a stereotype.

it will be never applicable. If it appears in a disjunction of the conclusion, then that disjunct should be eliminated from the dependency conclusion because it will never be possible to repair the constraint in the way it specifies.

5. Computing repairs

Once RGDs and eRGDs have been generated, we use them to compute the *minimal repairs* for a given set of structural events E to be applied in some information base IB. In other words, we compute the additional structural events R_i such that $apply(E \cup R_i, IB) \models C$, where C is the set of all the constraints defined in the schema. Intuitively, we only need to *chase* (i.e., to execute) the RGDs with the current IB and E and then, for each R_i obtained, check whether it is minimal or not. We define in this section the chase-like algorithms we use for computing such *repairs*. We start by showing how to chase the RGDs obtained from $\text{OCL}_{\mathsf{UNIV}}$ constraints using a current existing chase algorithm and then, we define how to chase the eRGDs obtained from OCL_{FO}.

5.1. Chasing RGDs from OCL_{UNIV}

In this particular case, we can make use of some preexisting chase algorithms to compute the *repairs*. Concretely, we show that we may use any *chase-like* algorithm that computes universal model sets to obtain the repairs R_i, provided that the algorithm supports dependencies with disjunctions and negative/built-in literals. Moreover, using such algorithms ensures that any returned R_i will be minimal.

We start from the definition of *universal model set*. Intuitively, given a set of ground facts I, a *universal model set* [33] is a minimal set of models $U \supseteq I$ (i.e., sets of ground facts that satisfy all the dependencies), possibly containing instances of a special kind of constants called *labeled nulls*, s.t. for any model $U' \supseteq I$, there is some model U in the universal model set and some substitution σ for the labeled nulls of U s.t. $U\sigma \subseteq U'$.

Consider now that we compute the universal model set for our set of RGDs obtained from $\text{OCL}_{\mathsf{UNIV}}$, using as ground facts the set $IB \cup E$. Clearly, any model U from the universal model set will correspond to the form $U = IB \cup E \cup R_i$, and will guarantee that U satisfies all the RGDs. In addition, U will not contain any labeled null since the unique sources of labeled nulls are the existential variables, but the RGDs drawn from $\text{OCL}_{\mathsf{UNIV}}$ are dependencies where all variables are universally quantified. Thus, any R_i from $U = IB \cup E \cup R_i$ is a repair.

Moreover, it will be minimal since, if there was some other (minimal) repair $R'_i \subset R_i$, then, $U' = IB \cup E \cup R'_i$ would also be a model from the universal model set, and since $U' \subset U$, the universal model set would not be minimal. Similarly, it can be argued that any R_j minimal repair will appear as some $U_j = IB \cup E \cup R_j$ in the universal model set, otherwise, there would not be any model U_i in the universal model set s.t. $U_i \subseteq U_j$.

A sound and complete algorithm for computing universal model sets for dependencies dealing with disjunctions, negated literals, and $\neq$ comparisons between terms is the extended core chase [33]. Unfortunately, this chase cannot deal with $<$ neither $\leq$ comparisons between terms, and both kind of inequalities may appear in our RGDs. However, it is not difficult to extend it to handle them when the variables compared are universally quantified, as it happens in our case, since it only requires checking the satisfaction of the comparison when applying a chase step, and rejecting the ongoing R_i computation if this is not the case.

5.2. Chasing e*RGDs from OCL_{FO}*

We cannot use currently existing chases when we have to deal with eRGDs coming from OCL_{FO} because their usual way of dealing with existential variables is to invent a new labeled null that may stand for either some new constant or some unknown constant currently existing in the information base IB or in the events E. This is not appropriate for us since we want our repairs to make explicit the existing values that can repair a violation.

That is, we are interested in computing repairs like $\iota takes(john, aspirin)$ instead of $\iota takes(john, null_1)$, since they provide a concrete way to repair the violation using the currently existing values in the IB.

To achieve this behavior, we need to define a *chase-like* procedure that incorporates the notion of Variable Instantiation Patterns (VIPs) [34]. Intuitively, VIPs rely on instantiating any existential variable with any current value from $IB \cup E \cup R_c$, where R_c is the current repair being computed, and some other new invented values. This chase procedure is defined in Algorithm 5.

Algorithm 5 chaseExtendedRGDs(*eRGDs*, *IB*, *E*, R_c, *Result*)

```
d := getViolatedDependency(RGDs, IB, E, R_c)
if d = null then
   Result.add(removeForbidContain(R_c))
else
   for all Structural events conjunction R in (d.conclusion) do
      σ_RS := getRepairingSubstitutions(R, IB, E, R_c)
      for all σ_R in σ_RS do
         chaseRGDs(eRGDs, IB, E, R_c ∪ {Rσ_R}, Result)
      end for
   end for
end if
```

Initially, the algorithm is called with $R_c = \emptyset$ and $Result = \emptyset$, where *Result* is an input/output parameter that will contain the set of all the repairs when the algorithm execution terminates. The *getViolatedDependency* function looks for a dependency being violated according to the contents of $IB \cup E \cup R_c$, and returns it after substituting its universal variables for the constants that witness the violation. If no dependency is violated, then R_c is a repair, and thus, it is added into the *Result*. Before adding R_c into *Results*, we apply the function *removeForbidContain*. This function returns the set of structural events R_c but removing those ground atoms corresponding to the auxiliary structural events $\iota forbidP$ and $\iota containP$ added as a consequence of the auxiliary dependencies created in Algorithm 4.

For repairing the dependency, we try all the possible conjunctions of events R in the dependency conclusion. Moreover, for each one of these conjunctions, we try all the suitable variable-to-constant substitutions σ_R for their existential variables according to the VIPs. Then, we add $R\sigma_R$ into the repair R_c being computed and apply a recursive call to the same algorithm to continue the chase until no dependency is violated.

After executing the algorithm, *Result* contain a set of repairs R_i for the given *IB* and *E*. Since the algorithm does not ensure their *minimality*, we must perform some additional check if we are interested in *minimal repairs*.

The algorithm is sound since it does not add any R_c as a repair in the *Results* unless it truly satisfies all the *e*RGDs. The algorithm is also complete since it is, in fact, a subcase of the CQC algorithm where unfoldings have been made explicit. The CQC algorithm has been proved to be complete with the VIPs approach, even in the presence of built-in literals such as $<$ and $\leq$ and considering both discrete and continuous domains for terms [34].

6. Identifying nice properties of OCL$_{\mathsf{UNIV}}$ Constraints

In addition to producing simpler repair generating dependencies, OCL$_{\mathsf{UNIV}}$ benefits from some nice properties that are interesting to investigate. Two of them are considered in this section: the assurance that the chasing algorithm ends when computing the repairs for OCL$_{\mathsf{UNIV}}$ constraints, and its relationship with OCL-Lite, a well-known decidable fragment of OCL [25].

6.1. Termination for computing repairs of OCL$_{\mathsf{UNIV}}$ constraints

We prove that computing OCL$_{\mathsf{UNIV}}$ constraints repairs will always terminate for any information base and for any set of structural events applied. The proof benefits from two observations: first, any OCL$_{\mathsf{UNIV}}$ constraint can be translated as a denial where all literals are base[3]. Second, any denial constraint where

[3]Generally, an OCL constraint is translated as a unique denial constraint. However, since any set of OCL constraints can be collapsed into a single one, for example, by using the `and` operator, an OCL constraint might be translated into two or more denial constraints.

all literals are base can be translated to an RGD where all variables from the conclusion are universally quantified. Thus, we prove termination by proving that *chasing* RGDs where all variables from the conclusion are universally quantified always terminates. We first prove the two observations.

Proposition 1. *Given a UML class diagram $\mathcal{U}$ and its logic formalization into base predicates $\mathcal{P}$, an OCL_{UNIV} constraint written over $\mathcal{U}$ can be translated into a set of denial constraints written over the set of base predicates $\mathcal{P}$.*

Proof. According to the translation from OCL to denial constrains as defined in [29], the unique OCL operators that generate negated literals when translated to denial constraints are `includes`, `includesAll`, `oclIsKindOf` and `exists`. The rest are translated using only positive literals, which can be iteratively unfolded until all positive literals are base. This is possible since the translation of [29] does not use recursive predicates.

For those operations using negative literals, we can see that they are translated using just one negative base literal due to our restricted syntax. Indeed, `includes` and `includesAll` are translated by means of a unique literal corresponding to the base predicate representing the association of the *role*. This is possible since the *source* of such operations is a single navigation step from some variable. Regarding `oclIsKindOf` and our limited `exists`, these operations are translated using one unique negated literal corresponding to the base predicate representing the *class* used by the operation. Thus, all negative literals are base.

Since positive and negative literals are base, no literal is derived and the claim is proven. □

Proposition 2. *Any denial constraint dc written over some set of base predicates $\mathcal{P}$ is translated as a set of different RGDs r where all variables in the conclusion are universally quantified, i.e., $r = \forall \overline{x}.\ \phi(\overline{x}) \rightarrow \bigvee \psi_i(\overline{x_i}), \overline{x_i} \subseteq \overline{x}$.*

Proof. We first show that any *dc* over some set of base predicates $\mathcal{P}$ is translated as a set of different safe EDCs, and from there, we show that each EDC gives rise to an RGD where all variables are universally quantified.

Any denial constraint *dc* is *safe*, so, any term in the formula appears at least in one positive literal. When translating *dc* to EDCs using the rules defined in 7, 8, any positive literal p is replaced by at least one positive literal with the same terms (p or ιp). Given that the translation to EDCs does not create new variables, the generated EDCs are also safe.

To obtain the RGDs from each of the previous EDCs, it is sufficient to move the negative literals from the premise to the conclusion of each rule (no unfolding is required since all literals are base). In this situation, we observe that all the terms of the conclusion of any generated RGD also appear in its premise (otherwise, the EDC would be unsafe). Thus, the generated RGDs are necessarily dependencies whose variables are universally quantified. □

Theorem 1. *Given a UML class diagram $\mathcal{U}$, and its logic formalization into base predicates $\mathcal{P}$, an OCL_{UNIV} constraint written over $\mathcal{U}$ can be translated into a set of RGDs R where all the variables of the RGD's conclusion are universally quantified, i.e., for any $r \in R, r$ has the form $\forall \overline{x}.\ \phi(\overline{x}) \rightarrow \bigvee \psi_i(\overline{x_i}), \overline{x_i} \subseteq \overline{x}$.*

Proof. Direct from Propositions 1 and 2. □

Theorem 2. *Given a UML class diagram $\mathcal{U}$, any information base IB, and any set of structural events E for $\mathcal{U}$, computing the repairs for a set C of OCL_{UNIV} invariants written over $\mathcal{U}$ by means of a chase terminates.*

Proof. From Theorem 1 we get to the conclusion that the set of $\text{OCL}_{\mathsf{UNIV}}$ constraints C is translated as a set of RGDs over some finite set of base predicates $\mathcal{P}$ where all variables from the conclusion are universally quantified.

Because the variables of all the RGDs are universal, the chase will not create new constants when creating new ground structural events, but will instead reuse those that already appear in IB and E, name them K. Since the set of predicates $\mathcal{P}$ encoding the class diagram is finite and the set of constants K used by the chase is finite, the number of different insertion/deletion events that can be defined with $\mathcal{P}$ and K is finite. Given that the chase stops when it does not instantiate a new different ground structural event and

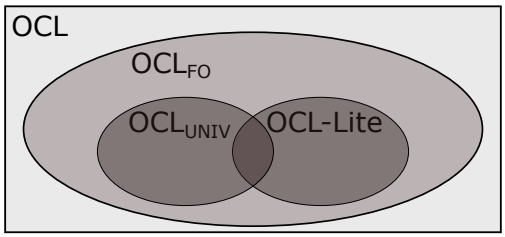

Figure 3: OCL$_{FO}$, OCL$_{UNIV}$ and OCL-Lite relations

it cannot keep creating new ground different structural events forever because they are finite, the chase will eventually terminate. □

6.2. Comparison of OCL$_{UNIV}$ with OCL-Lite

OCL-Lite is a decidable subset of OCL [25], meaning that, given a set of OCL-Lite invariants for a specific UML class diagram U, and an initial information base IB for U, we can correctly compute a new information base $IB' \supseteq IB$ for U satisfying every OCL-Lite invariant, or answer that no consistent IB' exists, in finite time.

It is easy to show that computing repairs for OCL-Lite invariants can also be done in finite time. Intuitively, given the initial information base IB, and some set of structural events E, we can compute in finite time a new $IB' \supseteq apply(E, IB)$ satisfying all the invariants due to the OCL-Lite decidability. Then, the repairs would be the structural events corresponding to the insertions of the instances $IB' \setminus apply(E, IB)$. In this way we can compute the repair that uses only insertion structural events. To compute repairs that include deletion structural events, we can compute such IB' not for $apply(E, IB)$, but for all $IB_s \subseteq apply(E, IB)$, name it IB'_s. Thus, the repairs are defined by the insertion structural events corresponding to $IB'_s \setminus apply(E, IB)$, and the deletion structural events corresponding to $apply(E, IB) \setminus IB'_s$.

Since computing repairs for OCL-Lite invariants can be done in finite time, our method will find those repairs also in finite time provided that the chase algorithm used is complete.

Given this situation, it is reasonable to ask ourselves whether OCL-Lite and OCL$_{UNIV}$ have any relationship in terms of expressiveness. In other words, can any constraint written in OCL$_{UNIV}$ be rewritten in OCL-Lite or viceversa? Our conclusion is that OCL-Lite and OCL$_{UNIV}$ are different languages (i.e., none of them subsumes the other), although they have a common intersection and both are subsets of the OCL$_{FO}$ fragment of OCL. This relationship is illustrated in Figure 3.

It is easy to see that some OCL-Lite invariants cannot be encoded in OCL$_{UNIV}$ since OCL-Lite has the `exists` operation unrestricted which would be translated into RGDs with existential variables in the conclusion. Such constraints cannot be encoded in OCL$_{UNIV}$. On the other hand, OCL$_{UNIV}$ has arithmetic comparisons, which are not allowed in OCL-Lite. As an example of the common part, both languages can use `forAll` in an unrestricted manner. To show that both are subsets of OCL$_{FO}$ it is sufficient to observe that all the operations that appear in OCL-Lite and OCL$_{UNIV}$ are in the syntax of OCL$_{FO}$ [26].

7. Experiments

To prove the feasibility of our approach and to analyze its efficiency, we have developed a prototype tool of our method and applied it to compute repairs in several situations related to a particular case study: the well-known EU-Car Rental UML/OCL schema [35]. We have considered two different scenarios to perform our experiments: the original version of the schema and also a simplified one, limited to constraints encodable in OCL$_{UNIV}$. We refer to the former as EU-Car Rental schema and the latter as EU-Car Rental OCL$_{UNIV}$ schema.

The goal of our experiments is to analyze the efficiency and the scalability of our approach, according to the following criteria:

- *Size of the initial information base.* We want to analyze to what extent our method scales up with the size of the information base.
- *Whether the applied structural events* do *or* do not *cause any violation.* We want to measure the time required for computing a repair when some constraint is violated, but also the time consumed when there is no constraint violation.
- *Whether the applied structural events are insertions or deletions.*
- *Whether the constraints considered are* OCL_{UNIV} *or not.*

We first describe the EU-Car Rental and EU-Car Rental $\text{OCL}_{\mathsf{UNIV}}$ schemas and then discuss the experiments performed with them.

7.1. EU-Car rental and EU-Car rental OCL_{UNIV} *schemas*

The EU-Car Rental system is aimed at specifying a fictional car rental company with the purpose of managing the *rentals* agreed, its *customers*, the rented *cars* and the different *branches* of the company, among other concepts.

We have used the EU-Car Rental schema that appears in [35] as the first schema for our case study. This schema has 21 classes/associations, 17 attributes and 74 explicit constraints (15 OCL constraints, 2 subtyping constraints and 57 min/max cardinalities).

The EU-Car Rental $\text{OCL}_{\mathsf{UNIV}}$ schema is the same as before but removing those constraints that cannot be encoded in $\text{OCL}_{\mathsf{UNIV}}$. In this particular example, we only had to remove minimum cardinality constraints since they were the only ones leading to dependencies with existential variables. Then, we came up with a total of 47 explicit constraints (15 OCL constraints, 2 subtyping constraints and 30 max. cardinalities).

7.2. Experiments

We implemented the translation algorithms in Java to obtain the repair-generating dependencies from the two EU Car Rental schemas. Then we chased the dependencies with some randomly generated information base and structural events. To perform the chase, we used a customized version of the SVTe tool [36]. Briefly, SVTe is a sound and complete reasoning engine developed in C# implementing the VIPs approach. For our purposes, we modified SVTe to behave as a chase. As a special feature, the input of SVTe is a set of EDCs rather than RGDs since SVTe internally interprets EDCs as RGDs. We performed all the experiments with an Intel Core i7-4710HQ up to 3.5Ghz, 8GB of RAM, running Windows 8.

First of all, we used our Java implementation of Algorithm 1 to translate all constraints of the EU-Car Rental and the EU-Car Rental $\text{OCL}_{\mathsf{UNIV}}$ schemas into EDCs. From this translation we obtained 437 EDCs and 393 EDCs in 2.58s and 2.71s respectively.

Then, we randomly built information bases of increasing size for which we applied two kinds of operations: one to create a new rental and another to delete a rental. Each scenario has been executed twice, one considering the EU Car Rental constraints and the other considering the EU Car Rental $\text{OCL}_{\mathsf{UNIV}}$ constraints.

The results of these experiments are shown in Tables 1 and 2. The first column in each table indicates the number of instances in the information base used. The following columns state the seconds taken to execute the chase until finding the first repair after taking out M necessary structural events from the set of events to be consistent in the EU-Car Rental Schema. That is, for the M=0 column, we used a set of structural events which ensured that no constraint was violated. For the other columns, we randomly took out $M = \{2, 4, 6, 8\}$ events in the initial set, thus causing some violation which required computing a repair. In this way, we ensured that all the violations were repairable, and controlled the size of the required repair.

To avoid the chase computing undesirable repairs (e.g. creating new branches of the company to pick-/drop some new rental), we restricted the events that can be used as repairs as explained in Section 4.4. In this way, when inserting new rentals, we limited the chase to repair constraints just by creating new rental instances, instances for its attributes and associations, and possibly new customers. For deletions, we allowed the chase to compute new rental deletions together with their corresponding associations/attributes.

Table 1 shows that our method scales well in the EU-Car Rental schema when inserting new rentals when no constraint is violated (case $M = 0$). As expected, the execution time increases when some violation

occurs and repairs need to be computed (cases $M > 0$). In this situation, the execution time increases with the size of the repair to be computed.

For deletions, our method takes about 2–3 min to compute repairs for an information base of 24,000 instances. Intuitively, these higher execution times are explained because, when some instance is deleted, we need to check in the information base whether some minimum cardinality is violated. This encompasses looking through all the instances of that association in the information base. This phenomenon turns out to be the bottleneck of the chase rather than the size of the repair to be computed since, as Table 2 shows, the execution times remain almost constant among the size of the repair needed.

On the other side, our method scales nicely when dealing with the version of the schema limited to OCL_{UNIV} for both cases, insertions and deletions. This can be explained because taking out structural events from the set of structural events to be applied especially violates minimum cardinality constraints, which are not encodable in OCL_{UNIV}.

For this reason, we decided to perform a new experiment with the EU-Car Rental OCL_{UNIV} schema with a different strategy for generating the structural events. In particular, we generated totally random structural events combining both insertions and deletions. We argue that this is the worst case since, when the structural events of an operation are randomly generated, the number of missing events to make it consistent (i.e., the number of structural events of the repair) may grow with each new structural event considered. Note that this case is just theoretical since operations are usually cohesive, and thus, not random.

In this experiment, most executions took less than one second, or just a few seconds (See Table 3). Execution times over 10s occurred with the largest IB of 24,000 instances (i.e., last row) and also with the largest N (i.e., last column values), especially when the size of the repair was composed of more than 20 structural events. Nevertheless, the maximum execution time was up to 1 min.

It is worth saying that we used a tool originally developed for satisfiability checking where only few instances needed to be taken in account. Thus, better results might be expected if considering a dedicated application with big data structure support.

Table 1: Execution time in seconds for new rental insertion

	EU-Car Rental					EU-Car Rental OCL_{UNIV}				
IB size	M=0	M=2	M=4	M=6	M=8	M=0	M=2	M=4	M=6	M=8
1168	0.11	0.11	0.14	0.78	0.58	0.11	0.10	0.12	0.90	0.09
1804	0.12	0.14	0.11	0.15	0.68	0.12	0.11	0.10	0.13	0.09
3525	0.14	0.12	0.36	0.60	1.96	0.14	0.10	0.13	0.11	0.10
6016	0.16	9.75	10.7	1.35	2.50	0.15	0.12	0.10	0.11	0.10
12567	0.21	0.27	0.24	14.6	8.29	0.20	0.21	0.18	0.14	0.12
24834	0.38	31.0	24.3	147	19.0	0.42	0.28	0.22	0.17	0.16

Table 2: Execution time in seconds for deleting a rental

	EU-Car Rental					EU-Car Rental OCL_{UNIV}				
IB size	M=0	M=2	M=4	M=6	M=8	M=0	M=2	M=4	M=6	M=8
1168	0.24	0.28	0.29	0.27	0.26	0.06	0.06	0.06	0.06	0.06
1804	0.61	0.56	0.51	0.57	0.56	0.06	0.06	0.06	0.06	0.06
3525	2.32	2.30	2.04	2.21	2.03	0.07	0.07	0.06	0.07	0.06
6016	7.10	5.98	7.10	7.49	5.73	0.15	0.07	0.07	0.07	0.07
12567	35.8	33.2	32.6	35.1	34.3	0.09	0.09	0.08	0.10	0.08
24834	161	141	187	146	147	0.12	0.10	0.12	0.11	0.11

8. Related work

To our knowledge, the unique work similar to ours is the OCLexec animation tool [19, 20]. Apart from it, there are several research areas that are closely related to ours: incremental constraint checking, model

Table 3: EU-Car Rental OCL_{UNIV} results for random insertions/deletions

	N=2		N=4		N=6		N=8	
IB size	Rep.	Time	Rep.	Time	Rep.	Time	Rep.	Time
1052	11	0.70	4	0.07	NR	0.09	NR	0.11
1877	5	0.14	2	0.15	NR	0.07	NR	0.10
3292	11	1.30	NR	0.09	NR	0.07	15	0.55
6539	3	0.12	NR	0.08	18	2.99	28	59.7
11739	3	0.61	6	6.51	NR	2.51	22	55.5
24272	3	14.1	0	0.04	11	19.2	NR	12.6

change propagation, and updates in Description Logics. We review them in this section.

8.1. OCLexec animation tool

Given an operation, the method proposed in [19, 20] translates its OCL postcondition and all the constraints defined in the conceptual schema into a SAT problem. Then, the method simulates the operation execution by means of invoking a SAT reasoner. Hence, the SAT reasoner returns an IB that is guaranteed to satisfy both the OCL operation postcondition together with all the constraints.

This approach benefits from the advantages of using well known SAT reasoners and it can be integrated in Java programs. Nevertheless, this approach cannot repair a constraint with class instance deletion. This implies that it cannot make use of instance *generalization* nor instance *subclass modification* (that is, removing or changing the type of a given instance) during maintenance. It is worth noting that they do not need to consider these cases since their work is focused on Java, where instances are not explicitly deleted and can not change their type. On the contrary, our method is not bound to any concrete technology and, thus, it considers also these kinds of repairs.

8.2. Incremental constraint checkers

By incremental constraint checkers we refer to those approaches that are able to detect in an incremental way whether an integrity constraint of the schema has been violated because of a change in the IB. Incrementality is achieved by evaluating only those constraints affected by the change and doing the evaluation only for the relevant data according to the change.

In [2], the author translates OCL constraints (limited to first-order logic constructs but with transitive closure) to graph patterns to benefit from incremental graph-pattern query algorithms. The work in [3] specifies the constraints in Prolog, and checks them using the updates of the elements that occur in the data. Proposals like [4, 5, 6] directly work with OCL interpreters from which they assess which OCL constraints should be checked and for which values.

It is worth noting that all these approaches are aimed at checking whether a constraint is violated but not to compute the repairs, as we do in this paper. Moreover, our method can also be used to check constraints incrementally directly using EDCs rather than its translation to RGDs. Indeed, an EDC is, by itself, a rule stating which kind of events causes a violation in a given information base *IB*. That is, it states the incremental checks that must be performed to ensure that no constraint is being violated because of an update. We have sketched how to use EDCs to perform incremental integrity checking in [37].

8.3. Model change propagation

There are several works concerned with propagating changes to models. That is, given some change applied in an element of a model diagram, compute the additional changes that need to be propagated to other model diagram elements to ensure their consistency. These approaches are working on the same problem of finding *repairs*, but in the concrete case in which the schema is a metamodel, the information base is the codification of one schema over that metamodel, and the constraints are the logic rules that should be true to ensure that the schema conforms to its metamodel.

In [15], the authors use an automated planning tool to compute the changes that should be applied to the information base to reach a consistent state. This approach is not incremental since it does not make use of the set of applied structural events E, but takes as input the new information-base state after its application (i.e., $apply(E, IB)$).

The approach in [16] extends its previous work in [5] to compute also the repairs in addition to incrementally checking violations of OCL constraints. Nevertheless, the work presented by the authors is intended to deal only with those *repairs* defined by a single structural event which, in turn, cannot contain any new constant (i.e., all constants have to be taken from IB or E).

The work presented in [14, 13] is based on compiling OCL constraints (limited to first-order constructs) into some production rules that generate *repairs*. The idea is that, when some violation occurs, the production rules are triggered, possibly violating new constraints, which might in turn trigger new production rules. Interestingly, this approach suggests assigning *cost* to the *repairs* (hence, preferred structural events will have lower cost); in this manner, the different *repairs* can be sorted before being shown to the user. However, the authors do not clarify how they deal with the case in which some production rule produces some new structural event that violates some previously repaired constraint. This is an important problem to tackle since it may cause an infinite loop.

Similarly, the work in [18] is intended to compute repairs for xlinkit's first-order logic constraints. It also works with production rules and, again, the method does not clarify how it deals with structural events that, although repairing some constraint, violate previously repaired constraints.

8.4. Updates in Description Logics

Description Logics research has also paid attention to the problem of consistently updating a *knowledge base* by means of updating its information base (i.e., the *ABox* in the knowledge base terminology). However, this research makes use of the *open world* interpretation of an information base which makes their concepts of *IB consistency* and *repair* to be different to ours.

Intuitively, an information base IB is *consistent* in the open world semantics if there exists some information base $IB_c \supseteq IB$ satisfying all the constraints defined, although IB might violate some of them. Since there might be several $IB_c \supseteq IB$ satisfying the constraints, it is useful to define the concept of logical consequences of IB, denoted by $cl(IB)$, as the intersection of all these IB_c.

Given an information base IB and some structural events E, the methods proposed in [10] and [11] are based on, first, computing the logical consequences $cl(IB)$, and then, the minimal deletion structural events R s.t. $apply(E \cup R, cl(IB))$ is consistent in terms of the open world semantics. In other words, the repair R is computed in such a manner to ensure the existence of some $IB_c \supseteq apply(E \cup R, cl(IB))$ satisfying all the constraints.

As a result, the set of additional structural events computed by these methods does not bring the IB to a new state satisfying all the constraints (closed world notion of consistency), but to a new IB that will satisfy all the constraints if some other unknown facts are properly inserted (open world notion of consistency).

Following our example about medicines, inserting in the IB the fact that some person *john* suffers *diabetes* without inserting any new medicine administration to *john* would bring a new consistent IB according to the open world semantics, and thus, no repair would be provided by the previous methods. In contrast, in the closed world semantics, the *AllSufferingDiseasesAreTreated* constraint would be violated, and thus, our method would compute a repair stating that we should add in the IB the fact that *john* is taking some medicine x, where x treats *diabetes*.

9. Conclusions

Computing the repairs that bring the data to a consistent state when a constraint has been violated is an important challenge in software development. On the one hand, it represents a step forward towards automatically enforcing constraints directly from the schema [21]. On the other hand, they are strictly necessary for maintaining the consistency of information systems. Our approach provides an important contribution in this direction, particularly for UML schemas with OCL constraints.

The proposed approach is able to compute repairs, that is, additional modifications of the data that avoid violating any integrity constraints for a certain update. We deal with constraints defined in OCL_{FO}, which is a rich fragment of OCL whose expressiveness is equivalent to relational algebra. We have also identified a particular fragment of OCL_{FO}, called $\text{OCL}_{\mathsf{UNIV}}$, for which we have proved termination of the repair computation process. Some techniques to reduce the number of obtained repairs have also been discussed.

As an additional application of our approach, we have shown how it can also be used to fix up non-executable operations in UML/OCL conceptual schemas.

We have conducted some experiments to analyze the scalability of our approach in practice, both for repairing constraints defined in $\text{OCL}_{\mathsf{UNIV}}$ and OCL_{FO}. We have shown that our approach performs efficiently with $\text{OCL}_{\mathsf{UNIV}}$ constraints and it also provides good results when dealing with OCL_{FO} constraints.

This work can be extended in several directions. First, by providing additional insight into the problem of reducing the number of repairs obtained. Second, by still going beyond the expressive power provided by OCL_{FO}. Third, by providing automatic translations to our logic formalization from other languages beyond UML/OCL, such as SQL or ORM for instance.

Acknowledgments

This work has been partly supported by the Ministerio de Economía y Competitividad under the projects TIN2014-52938-C2-2-R, and TIN2011-24747, and by the Secretaria d'Universitats i Recerca de la Generalitat de Catalunya under 2014 SGR 1534 and an FI grant.

References

[1] J. Van Griethuysen, Concepts and terminology for the conceptual schema and the information base, ISO/TC97/SC5/WG3, ISO/TC97 Computers and Information Processing, 1982.

[2] G. Bergmann, Translating OCL to graph patterns, in: Model-Driven Engineering Languages and Systems, Vol. 8767 of LNCS, Springer, 2014, pp. 670–686.

[3] J. R. Falleri, X. Blanc, R. Bendraou, M. A. Almeida Da Silva, C. Teyton, Incremental inconsistency detection with low memory overhead, Software: Practice and Experience 44 (5) (2014) 621–641.

[4] A. Uhl, T. Goldschmidt, M. Holzleitner, Using an OCL impact analysis algorithm for view-based textual modelling, ECEASST 44 (2011) 1–20.

[5] I. Groher, A. Reder, A. Egyed, Incremental consistency checking of dynamic constraints, in: Fundamental Approaches to Software Engineering, Springer, 2010, pp. 203–217.

[6] J. Cabot, E. Teniente, Incremental integrity checking of UML/OCL conceptual schemas, Journal of Systems and Software 82 (9) (2009) 1459–1478.

[7] J. Rumbaugh, I. Jacobson, G. Booch, Unified Modeling Language Reference Manual, 2nd Edition, Pearson Education, 2005.

[8] Object Management Group (OMG), Unified Modeling Language (UML) Superstructure Specification, version 2.4.1, `http://www.omg.org/spec/UML/` (2011).

[9] Object Management Group (OMG), Object Constraint Language (UML), version 2.4, `http://www.omg.org/spec/OCL/` (2014).

[10] M. Lenzerini, D. F. Savo, Updating inconsistent description logic knowledge bases., Vol. 242 of Frontiers in Artificial Intelligence and Applications, 2012, pp. 516–521.

[11] D. Calvanese, E. Kharlamov, W. Nutt, D. Zheleznyakov, Updating ABoxes in DL-Lite, in: Alberto Mendelzon Workshop on Foundations of Data Management (AMW), Argentina, 2010.

[12] R. Reiter, On closed world data bases, in: H. Gallaire, J. Minker (Eds.), Logic and Data Bases, Springer US, 1978, pp. 55–76.

[13] H. K. Dam, M. Winikoff, Supporting change propagation in UML models, in: Software Maintenance (ICSM), 2010 IEEE International Conference on, 2010, pp. 1–10.

[14] K. Dam, M. Winikoff, Generation of repair plans for change propagation, in: M. Luck, L. Padgham (Eds.), Agent-Oriented Software Engineering VIII, Vol. 4951 of LNCS, Springer, 2008, pp. 132–146.

[15] J. Pinna Puissant, R. Van Der Straeten, T. Mens, Badger: A regression planner to resolve design model inconsistencies, in: Modelling Foundations and Applications, Vol. 7349 of LNCS, Springer, 2012, pp. 146–161.

[16] A. Egyed, E. Letier, A. Finkelstein, Generating and evaluating choices for fixing inconsistencies in UML design models, in: Automated Software Engineering, 2008. ASE 2008, 2008, pp. 99–108.

[17] C. Nentwich, W. Emmerich, A. Finkelsteiin, E. Ellmer, Flexible consistency checking, ACM Trans. Softw. Eng. Methodol. 12 (1) (2003) 28–63.

[18] C. Nentwich, W. Emmerich, A. Finkelstein, Consistency management with repair actions, in: Software Engineering, 2003. Proceedings. 25th International Conference on, 2003, pp. 455–464.
[19] M. P. Krieger, A. Knapp, B. Wolff, Automatic and efficient simulation of operation contracts, in: Proceedings of the 9th International Conference on Generative Programming and Component Engineering, GPCE '10, ACM, New York, NY, USA, 2010, pp. 53–62.
[20] M. P. Krieger, A. Knapp, Executing underspecified OCL operation contracts with a SAT solver, Electronic Communications of the EASST 15 (2008) 1–17.
[21] A. Olivé, Conceptual schema-centric development: A grand challenge for information systems research, in: Advanced Information Systems Engineering, Vol. 3520 of LNCS, Springer, 2005, pp. 1–15.
[22] D. W. Embley, S. W. Liddle, O. Pastor, Conceptual-model programming: A manifesto, in: D. W. Embley, B. Thalheim (Eds.), Handbook of Conceptual Modeling, Springer, 2011, pp. 3–16.
[23] X. Oriol, E. Teniente, A. Tort, Fixing up non-executable operations in UML/OCL conceptual schemas, in: Conceptual Modeling, Vol. 8824 of LNCS, Springer, 2014, pp. 232–245.
[24] A. Queralt, E. Teniente, Specifying the semantics of operation contracts in conceptual modeling, in: Journal on Data Semantics VII, Vol. 4244 of LNCS, Springer, 2006, pp. 33–56.
[25] A. Queralt, A. Artale, D. Calvanese, E. Teniente, OCL-Lite: Finite reasoning on UML/OCL conceptual schemas, Data & Knowledge Engineering 73 (0) (2012) 1 – 22.
[26] E. Franconi, A. Mosca, X. Oriol, G. Rull, E. Teniente, Logic foundations of the OCL modelling language, in: Logics in Artificial Intelligence, Vol. 8761 of LNCS, Springer, 2014, pp. 657–664.
[27] J. W. Lloyd, Foundations of Logic Programming, 2nd Edition, Springer Verlag, 1987.
[28] A. Olivé, Conceptual Modeling of Information Systems, Springer, Berlin, 2007.
[29] A. Queralt, E. Teniente, Verification and validation of UML conceptual schemas with OCL constraints, ACM TOSEM 21 (2) (2012) 13.
[30] A. Queralt, E. Teniente, Reasoning on UML conceptual schemas with operations, in: 21st International Conference on Advanced Information Systems Engineering (CAiSE'09), Vol. 5565, Springer, 2009, pp. 47–62.
[31] A. Olivé, Integrity constraints checking in deductive databases, in: Proceedings of the 17th Int. Conference on Very Large Data Bases (VLDB), 1991, pp. 513–523.
[32] G. Mecca, G. Rull, D. Santoro, E. Teniente, Ontology-based mappings, Data & Knowledge Engineering (to appear).
[33] A. Deutsch, A. Nash, J. Remmel, The chase revisited, in: Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '08, ACM, New York, USA, 2008, pp. 149–158.
[34] C. Farré, E. Teniente, T. Urpí, Checking query containment with the CQC method, Data & Knowledge Engineering 53 (2) (2005) 163–223.
[35] D. Costal, C. Gómez, A. Queralt, R. Raventós, E. Teniente, Improving the definition of general constraints in UML, Software & Systems Modeling 7 (4) (2008) 469–486.
[36] C. Farré, G. Rull, E. Teniente, T. Urpí, SVTe: a tool to validate database schemas giving explanations, in: 1st Int. Workshop on Testing database systems, DBTest '08, ACM, New York, NY, USA, 2008, pp. 9:1–9:6.
[37] X. Oriol, E. Teniente, Incremental checking of OCL constraints through SQL queries, in: Proceedings of the 14th International Workshop on OCL and Textual Modelling, 2014, pp. 23–32.