Simplification of UML/OCL Schemas for Efficient Reasoning

Xavier Oriol, Ernest Teniente

Universitat Politècnica de Catalunya – Barcelona, Spain

Abstract

Ensuring the correctness of a conceptual schema is an essential task in order to avoid the propagation of errors during software development. The kind of reasoning required to perform such task is known to be exponential for UML class diagrams alone and even harder when considering OCL constraints. Motivated by this issue, we propose an innovative method aimed at removing constraints and other UML elements of the schema to obtain a simplified one that preserve the same reasoning outcomes. In this way, we can reason about the correctness of the initial artifact by reasoning on a simplified version of it. Thus, the efficiency of the reasoning process is significantly improved. In addition, since our method is independent from the reasoning engine used, any reasoning method may benefit from it.

Keywords: UML, OCL, reasoning, simplification

1. Introduction

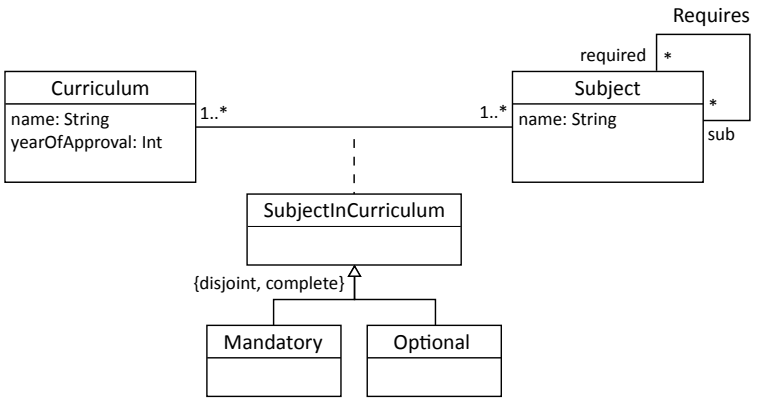
A conceptual schema is the description of an information system in terms of the data that it contains and the operations it provides to modify that data. According to the principle of necessity [1], the development of an information system always encompasses the definition of its conceptual schema. Moreover, methodologies like Model-Driven Development require the conceptual schema to be formally documented using well standardized languages.

Probably, the most well-known modeling languages for specifying conceptual schemas are UML and OCL [2, 3], which are standards maintained by the OMG [4, 5]. UML allows defining class diagrams (i.e., a taxonomy of classes and the associations between them), while OCL is used to formally define textual constraints over the class diagram (i.e., conditions that the instances of the class diagram must always satisfy). We use the name *UML/OCL schema* to refer to a UML class diagram annotated with OCL textual constraints.

For instance, consider the UML/OCL schema in Figure 1. This schema specifies an information system storing the *curriculums* offered by some university. That is, for each *curriculum*, it stores its *subjects* distinguishing whether they are *mandatory* or *optional*. Furthermore, the system stores the subjects *required* to be passed before enrolling some other subject. This schema is complemented with four OCL constraints. The first two state that the primary key of both the curriculums and the subjects is its name. *RequiredSubjectBelongsToCurriculum* ensures that if some subject is included in a curriculum, every subject required by it is also in the curriculum. Finally, *MandatoryRequirementsAreMandatory* forces all subjects required by a mandatory subject of a curriculum to be mandatory in the same curriculum.

It is well known that the mistakes made during the specification of an information system, and thus, its conceptual schema, are the most frequent mistakes (Boehm's first law [6]). Moreover, fixing them is more expensive the later in the software development process they are found and addressed, hence, it is necessary to assess the correctness of a conceptual schema as early as possible.

Email addresses: xoriol@essi.upc.edu (Xavier Oriol), teniente@essi.upc.edu (Ernest Teniente)




```
context Curriculum inv CurriculumKey:
Curriculum.allInstances()->isUnique(name)

context Subject inv SubjectKey:
Subject.allInstances()->isUnique(name)

context Curriculum inv RequiredSubjectsBelongsToCurriculum:
self.subject->includesAll(self.subject.required)

context Mandatory inv MandatoryRequirementsAreMandatory:
self.subject.required.subjectInCurriculum->forAll(s|
    s.curriculum = self.curriculum implies s.oclIsKindOf(Mandatory))
```

Figure 1: UML/OCL schema for the studies programs offered by a university

Checking the correctness of a schema consists in ensuring that it satisfies a set of desirable properties. For instance, one of the most classical properties to ensure is class *liveliness* (aka *satisfiability* or *consistency*). A class C is said to be lively if there is an instantiation of the class diagram satisfying all the constraints and containing at least one instance of C. Other properties to check the correctness of a schema have been defined in the literature such as non-redundancy of constraints, state reachability, etc.[7].

To (semi)automatically check these properties, several methods and reasoning tools have been proposed so far. See, for instance, UMLtoCSP [8], USE[9], AuRUS[10] or UML2Alloy [11]. Unfortunately, since checking these properties is exponential with the UML class diagram alone [12], and the addition of OCL constraints increases its complexity, efficiency is currently one of the major issues of such tools [13]. Taking into account that industrial/large schemas may easily contain more than 100 classes [14], improving the performance of these tools becomes a necessary requirement to adopt them in software production.

In this paper, we address the problem of improving the efficiency of these reasoning tools by means of *simplifiying* the UML/OCL schema they reason with. Our main observation is that, given a property to test, only some elements of the UML/OCL schema are relevant to determine its satisfaction. Intuitively, these elements are those constraints that might *contradict* the property, and the classes, associations, and attributes referred by these constraints. Then, the rest of the elements (i.e., attributes, classes, associations, and/or constraints) may be removed from the schema without altering the outcomes of checking the property. Clearly, removing these elements improves the performance of the reasoning tool since it allows abstracting from irrelevant elements and focus only on the significant ones while reasoning.

We call *simplification* to the process of detecting and removing the irrelevant elements of some property to check, and *simplified schema* to the schema obtained after it. In some cases, the simplification process might remove all the elements of the schema, thus, obtaining the empty simplified schema. This situation occurs when there is no element of the original schema that might contradict the property, which means that the property is directly satisfied and no reasoning is needed to check it.

The simplification process we propose in this paper is fully based on syntactic criteria, thus, ensuring good execution times for simplification; and it is powerful enough to obtain empty simplified schemas, i.e., it is even capable sometimes to assess the satisfaction of some properties without using any reasoning engine.

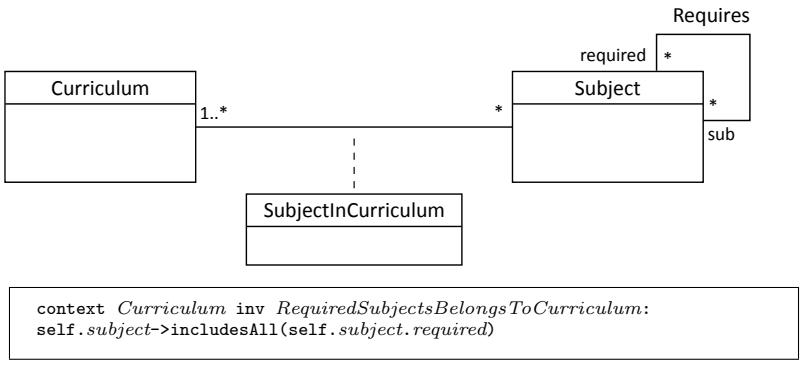


Figure 2: Simplified UML/OCL schema to check whether *Requires* is lively

For instance, given the UML/OCL schema in Figure 1, our simplification technique is able to directly assess that the class *Curriculum* is lively since we get the empty schema after simplifying it. On the other hand, when checking the liveliness of the association *Requires*, we obtain the significantly simpler schema shown in Figure 2. In particular, note that the simplification process has removed all the attributes (with their corresponding attribute min/max cardinalities), two UML classes (and its hierarchy relationships), an association minimum cardinality, the disjoint/completeness constraints, and three OCL constraints. Thus, a total of 14 out of 24 constraints can be removed from the original schema considering UML and OCL constraints.

Note that, since the simplification approach works directly on the UML/OCL schema, any reasoning engine can benefit from it. To illustrate this, and to analyze the efficiency improvement we achieve, we have performed some experiments combining different tools and schemas. In particular, we have checked several properties on the DBLP [15] and the EU-Car rental[16] schemas using two different reasoning tools, USE[17], and AuRUS[10], considering both, its simplified and its original schema versions. In these experiments, we show that the improvements in performance gained with our proposal range from x2 to x370 faster. Taking in account that the simplification process consumed negligible time (up to 0.5 seconds in the worst case), we believe that our technique can be adopted by developers to improve the efficiency of their tools. It is worth mentioning that improving the efficiency of these tools is a must for scaling them for industrial purposes.

We summarize the main contributions of this paper as follows: (1) we formally define the concept of schema simplification, (2) we develop an efficient method to compute simplified schemas and prove its correctness, and (3) we experimentally show the benefits of our method regarding execution time performance in different reasoning tools. Moreover, since all our approach is based on a logic formalization of the UML/OCL schema, it can also be applied to any other conceptual modeling language that might be encoded in this formalization. It is worth saying that the idea of removing irrelevant elements from UML/OCL schemas to enhance reasoning execution times is already present, and informally developed, in several reasoning techniques [18, 19, 20, 8]. However, as far as we know, we are the first ones to formalize the notions of simplification, bringing mathematical bases and proofs, and empirical evidence of its benefits. Thus, in this paper we address an actual gap found in the conceptual modeling reasoning field.

The rest of the paper is organized as follows. First, in Section 2, we review some logic notions we will use throughout the paper. Then, we formally define the concept of simplification and sketch our simplification proposal in Section 3. Afterwards, we develop the method in three different steps explained in Sections 4, 5, and 6. The experiment results and related work are discussed in Sections 7 and 8, respectively. Finally, in Section 9 we present our conclusions and outline future work.

2. Background and Notation

In this section, we overview some logic background and notation used along all the paper. Our approach is based on a classical 2-valued logic semantics, which does not correspond to the OCL standard. However,

with regards to reasoning in OCL, most OCL reasoning tools also assume a 2-valued logic semantics (like USE-Kodkod [17], AuRUS [21] or UMLtoCSP [8]). Indeed, the kind of reasoning task involved can be perfectly emulated with a 2-valued logic. This is because, at the end, these reasoners try to build an instantiation of the schema where all constraints evaluate to true (not to false, nor null, nor invalid). Thus, the reasoning tools can collapse the null/invalid values to false. Since our aim is to improve the performance of such tools, we can use their same 2-valued logic semantics for the same reasons.

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 rule of the form:

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

Where $\overline{t_h} \subseteq \bar{t}$. In the formula, $p(\overline{t_h})$ 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 its head, $p(\bar{t})$ is evaluated to true if and only if one of the bodies of such derivation rules is evaluated to true.

We extend the notion of base/derived predicates to atoms and literals. That is, when the predicate of some atom/literal is base, we say that such atom/literal is base too, otherwise, we say that it is derived.

Instance, and instantiation A ground atom of some base predicate p is called an *instance* of p. Then, a finite set I of instances of one or more predicates is called an instantiation.

Substitution A *substitution* θ is a set of the form $\{x_1/t_1, ..., x_n/t_n\}$ where each variable x_i is unique. The domain of a substitution is the set of all x_i and is referred as $dom(\theta)$. We say that θ is ground if every t_i is a constant. The literal $l\theta$ is the literal resulting from simultaneously substituting any occurrence of x_i in l for its corresponding t_i. Similarly, we define the conjunction $\phi\theta$ as the conjunction resulting from simultaneously applying the substitution θ to all the literals of ϕ. We say that a substitution θ is *valid* for a conjunction of literals ϕ if and only if $\phi\theta$ does not encompass a contradiction with the built-in literals of ϕ (e.g. $\{x/y\}$ is a non-valid substitution for $p(x, y) \wedge x \neq y$).

Denial constraints A *denial constraint* is a rule of the form:

$$\forall \bar{t}.\ \phi(\bar{t}) \rightarrow \bot$$

Where ϕ is a conjunction of (possibly derived) literals and $\bot$ is an atom that evaluates to false. We suppose all denial constraints to be safe (i.e., each variable appearing in a negated or built-in literal also appears in a positive literal). Intuitively, the left hand side (LHS) of a denial constraint express a condition that should never be satisfied by an instantiation.

Disjunctive embedded dependencies A *disjunctive embedded dependency* (*ded*) is a rule of the form:

$$\forall \bar{t}.\ \phi(\overline{t_\phi}) \rightarrow \bigvee_{i=1..n} \exists \overline{y_i}.\ \psi_i(\overline{t_i}, \overline{y_i})$$

where all literals are positive and base. It is important to hightlight that n might be 0, and thus, the right-hand side might be empty. In such case, we use the convention that the empty disjunction evaluates to false [22] and write $\bot$ to represent so. Note that *deds* are a kind of tuple-generating dependencies allowing disjunctions in the right hand side.

Ded violation/repair Given some instantiation I, we say that I violates a *ded* if I satisfies the LHS of the *ded* (i.e., there exists some ground substitution θ s.t. $\phi(\overline{t_\phi})\theta \subseteq I$) but I does not satisfy the RHS of the same *ded* (i.e., there does not exist any ground substitution σ s.t. for some ψ_i, $\psi_i(\overline{t_i}, \overline{y_i})\theta\sigma \subseteq I$). We use the usual symbol $\models$ to express that I satisfies some formula or *ded*.

When some I violates some *ded*, *repairing* the *ded* means building a new set $I' \supset I$ including the necessary instances of predicates of the RHS s.t. for any ground substitution θ, if $I' \models \phi(\overline{t_\phi})\theta$, then, $I' \models \psi_i(\overline{t_i}, \overline{y_i})\theta$ for some ψ_i.

Consistency, and satisfiability An instantiation I is said to be *consistent* with respect to a set of *deds* D (written as $I \models D$), if it does not violate any $d \in D$. A set of *deds* D is said to be *satisfiable* if there exists some consistent instantiation I for it. Equivalently, a set of *deds* is satisfiable if and only if it is possible to repair an empty instantiation $I_0 = \emptyset$ with respect to D.

3. UML/OCL Schema Simplification

Given a UML/OCL schema and a property to test, simplifying the schema consists in removing attributes, classes, associations, and constraints from the schema without altering the satisfiability of the property.

To simplify a UML/OCL schema, we rely on a logic formalization of the schema that allows us to endow it with a precise semantics, and so that we can also formally define the property to test. We use the proposal in [10] with this purpose where a UML/OCL schema is specified as a set of first-order constraints and the property to test as a satisfiability checking problem on the logic formalization of the schema. More precisely, the predicates in the logic correspond to the names of the classes, associations, and attributes of the schema, while the formulas are obtained from the UML and OCL constraints.

Given a logic formalization S of a UML/OCL schema, consider a first-order formula *goalToSatisfy()* corresponding to the property we want to test. For instance, the *goalToSatisfy()* of the Curriculum liveliness reasoning task is *Curriculum(c)*. Note that more difficult *goalToSatisfy()* formulas could be defined, e.g. *Curriculum(c)* $\wedge$ *SubjectInCurriculum(sc, c, s)* $\wedge\neg$*Mandatory(sc)* would be a *goalToSatisfy()* to test if there can exists a Curriculum with some subject which is not mandatory.

Then, we can formally define the simplified schema S' of S as follows:

Definition 1. *S' is a simplified schema of S for* goalToSatisfy() *if and only if: (1) $S' \subset S$, and (2)* {goalToSatisfy()}$\cup S'$ *is satisfiable iff* {goalToSatisfy()}$\cup S$ *is satisfiable.*

Intuitively, we can remove from S all predicates that do not appear in any logic constraint nor in *goalToSatisfy()* without altering its satisfiability. Equivalently, we can remove from the original UML/OCL schema all classes, attributes and associations not referred by any constraint neither by the property to test. Hence, our main target is to delete as many constraints as possible, since deleting constraints allows to remove other elements from the schema.

It is worth noting that we aim at obtaining a *proper* subset of the original schema but not the *minimal* one. This is because computing a minimally simplified schema might be so costly, that its reasoning execution time improvement might not be worthwhile. In fact, as we show in the following theorem, computing the *minimal* simplification of S for some *goalToSatisfy()* is as difficult as checking the satisfiability of *goalToSatisfy()* itself.

Theorem 1. *Checking the satisfiability of* $S\cup$\{goalToSatisfy()\} *can be reduced to computing the minimal simplification S' of S for* goalToSatisfy(), *that is, computing the subset $S' \subset S$ s.t. there is no other simplified schema S'' for S s.t. $S'' \subset S'$.*

Proof. We reduce the problem of checking whether $S\cup\{$*goalToSatisfy()*$\}$ is satisfiable to checking whether the minimal simplified schema is the empty schema. Indeed, if $S\cup\{$*goalToSatisfy()*$\}$ is not satisfiable, its minimal simplification is not the empty schema since the empty schema is trivially satisfiable. On the contrary, if $S\cup\{$*goalToSatisfy()*$\}$ is satisfiable, the empty schema is its minimal simplified schema since it is also satisfiable and it has no proper subset. □

Thus, our goal is to find a good tradeoff between the cost to compute the simplified schema and the benefit obtained from reasoning on it. Therefore, we aim at keeping low the cost of computing the simplified schema by considering only a syntactic analysis of the schema that can be efficiently checked. Then, since testing some property in a schema is inherently exponential, just a linear reduction on the size of the schema might potentially outcome a benefit of orders of magnitude. In the following, we overview this proposal.

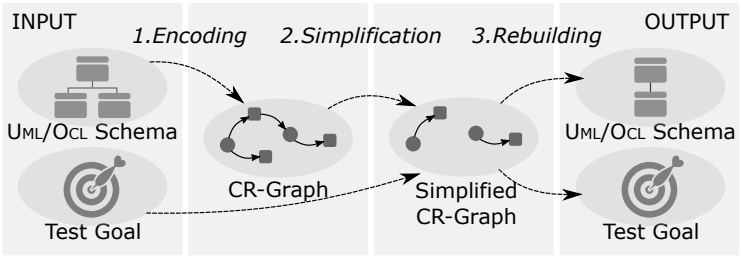


Figure 3: Overview of the method

3.1. Our Simplification Proposal In a Nutshell

Given a logic formalization S of a UML/OCL schema, checking whether a formula *goalToSatisfy()* is satisfiable on S consists in identifying an instantiation I satisfying both the constraints in S and *goalToSatisfy()*. Thus, our goal is to distinguish between the constraints in S that may make *goalToSatisfy()* unsatisfiable from those that cannot. So, the later constraints can be removed from the schema without altering the satisfiability of the goal.

To identify the constraints that may make *goalToSatisfy()* unsatisfiable, we exploit the fact that checking whether some goal is satisfiable on S is equivalent to check whether we can repair an instantiation satisfying the goal. Indeed, consider some instantiation I_1 satisfying the goal. I_1 might not witness the satisfiability of the goal if it violates some constraint. However, we might be able to add a sufficient (finite) amount of new instances into I_1 to repair such constraints while still satisfying the goal. Nevertheless, the new instances added may induce the violation of other constraints. This forms an iterative process that is applied until we reach a consistent instantiation I_n, or we reach an inconsistent instantiation I_m that cannot be repaired. The first case means that the goal is satisfiable, while the second means that it is unsatisfiable.

In this way, it is easy to see that the constraints that might make the goal unsatisfiable are those constraints that (1) can be violated when trying to satisfy the goal (or repairing its subsequently violated constraints), and (2) can violate some other constraint when repaired.

To illustrate so, assume that our goal is to find an instantiation to prove that *Curriculum* is lively. An instantiation containing only one instance of *Curriculum* violates, among others, the minimum cardinality constraint that states that a curriculum requires one *Subject*. Then, we need to consider an additional instance of *Subject* related to the previous instance of curriculum. However, this instance of the class *SubjectInCurriculum* will violate the completeness constraint of its hierarchy. Hence, this minimum cardinaliaty constraint might make *Curriculum* unsatisfiable since it can be violated when trying to satisfy the goal, and can violate another constraint when repaired. However, consider now the completeness hierarchy constraint. This constraint, when violated, can always be repaired by means of making the *SubjectInCurriculum* instance to be instance of the *Optional* subclass. Moreover, this *repair* does not violate any other constraint. Thus, this *complete hierarchy* constraint cannot make *Curriculum* unsatisfiable.

We build a constraint-repair graph (*CR-Graph*, for short) to be able to analyze this interaction of constraints/repairs so that we can identify the constraints in the schema that may lead to the unsatisfiability of the goal. Intuitively, the vertices of the CR-graph represent the constraints and the goal to satisfy while the edges depict which constraints can be violated when repairing another constraint or when satisfying the goal. From the CR-graph we can identify the constraints that are not responsible for the unsatisfiability of the goal which will be consequently removed from it.

In Figure 3 we show all the steps we take to simplify a UML/OCL schema. Briefly, each step behaves as follows:

- **CR-Graph encoding.** We first encode the UML/OCL schema into a CR-graph. This is done by means of translating the UML/OCL schema into a set of *disjunctive embedded dependencies* (*deds*), and then depict these *deds* as vertices in the graph.

- **CR-Graph simplfication.** Then, we add the goal to satisfy as a new *ded* in the graph, and we proceed to its simplification. That is, we remove some of the *deds*, i.e., vertices from the graph, while preserving the satisfiability of the goal.

- **Drawing the simplified UML/OCL schema.** Finally, we build the new, simplified, UML/OCL schema from the remaining *deds* of the graph. That is, we rebuild a UML/OCL schema whose logic representation corresponds to the remaining *deds* from the graph.

For drawing the simplified UML/OCL schema, we might need to access again the original UML/OCL schema, so, a copy of the original UML/OCL schema is required until the end of the process. Moreover, we need to keep track of which *deds* correspond to which UML/OCL constraints, so that, when removing the deds, we know which UML/OCL schema constraints are deleted.

4. Encoding the UML/OCL Schema into a CR-graph

First of all, we encode the original UML/OCL schema into a CR-graph. Such graph will be used later for identifying the constraints that can be deleted. Briefly, we obtain this graph in two steps:

1. Translating the UML and OCL constraints into *deds*.
2. Building the CR-graph from the *deds*.

In the following, we present both steps separately.

4.1. Translating the UML/OCL Schema into Deds

We obtain the *deds* required to encode the constraints in the UML/OCL schema by first translating the constraints into *denial constraints* as defined in [10], and then rewritting these denials into *deds* according to [23]. In the following, we briefly illustrate both techniques for the self-containment of the paper.

4.1.1. Translating the UML/OCL Constraints into Denial Constraints

First of all, we obtain the logic formalization of the UML/OCL schema. This is achieved by considering a different predicate for each class, attribute, and association in it. For instance, our schema example in Figure 1 brings the following predicates:

```
curriculum(c), curriculumName(c, n), curriculumYear(c, y),  subject(s), subjectName(s, n),
        requires(s1, s2), subjInCurriculum(sc, s, c), mandatory(sc), optional(sc)
```

Now, *RequiredSubjectsBelongsToCurriculum* can be encoded as the denial:

$$\texttt{requires(s,s2)} \land \texttt{subjInCurriculum(sc,s,c)} \land \neg\texttt{subjInCurriculum(sc2,s2,c)} \rightarrow \bot$$ [1]

The previous formula states that there is a violation of the constraint (stated by the $\bot$ symbol) if there is a subject s requiring another subject $s2$ so that s belongs to a curriculum c but $s2$ does not. We have omitted the logic quantifiers in the formula since they can be understood from the context.

In its simplest form, denial constraints are only defined by means of base literals like the previous example. However they might contain derived literals in the general case. This is shown in the following denial, which is the encoding of the *complete* hierarchy constraint of the class *SubjectInCurriculum*:

$$\texttt{subjInCurriculum(sc,s,c)} \land \neg\texttt{subjInCurriculumIsKindOf(sc)} \rightarrow \bot$$
$$\texttt{subjInCurriculumIsKindOf(sc)} \leftarrow \texttt{mandatory(sc)}$$
$$\texttt{subjInCurriculumIsKindOf(sc)} \leftarrow \texttt{optional(sc)}$$

Note that *subjInCurriculumIsKindOf(sc)* is a derived literal that evaluates to true if and only if *mandatory(sc)* or *optional(sc)* does. In this way, the first rule is a denial stating that if *sc* is a subject in some curriculum and it is not *mandatory* neither *optional*, then the constraint is violated.

[1] For the sake of readability, we present this formula with an *unsafe* term *sc2*. Such term can be easily made safe using a new derived predicate.

Following this encoding we can deal with most typical UML constructs: classes, attributes, associations, association classes, n-ary associations, compositions, aggregations, and enumerations. However, the encoding does not distinguish between different UML datatypes. Consequently, specific datatype operations such as `substring` are not supported.

With regards to UML/OCL constraints, the denials with (non-recursive) derived literals we use are expressive enough to encode:

- *UML constraints:* i.e., referential integrity constraints of associations, identifiers of association classes, min/maximum cardinality constraints of attributes/associations, hierarchies, disjointness and completeness integrity constraints.
- *OCL_{FO} constraints:* i.e., OCL constraints limited to first-order constructs. That is, OCL constraints not using transitive closure neither general aggregates such as `sum()` (or operations that could emulate them such as `iterate`). The full syntax of this language can be found in [24].

4.1.2. Rewriting Denial Constraints into Deds

Denial constraints are rewritten into *deds* by moving the negated literals from its left-hand side (LHS) of the formula to its right-hand side (RHS) and making them positive. Hence, the denial we have just seen for *RequiredSubjectsBelongsToCurriculum* leads to:

`requires(s,s2)` $\wedge$ `subjInCurriculum(sc,s,c)` $\rightarrow$ `subjInCurriculum(sc2,s2,c)`

This *ded* states that *if* there is a subject s requiring another subject $s2$ and s belongs to the curriculum c, *then*, $s2$ must also belong to c.

The previous simple rewriting is sufficient when the denial constraint has no derived literals. However, in the presence of derived literals, we need to apply an additional unfolding step, as it happens with the denial encoding the *complete* hierarchy constraint of the class *SubjectInCurriculum*. In this case, when we move the negated literal into the RHS we obtain:

`subjInCurriculum(sc,s,c)` $\rightarrow$ `subjInCurriculumIsKindOf(sc)`

Now, *subjInCurriculumIsKindOf* must be replaced by its unfolding, i.e., by the disjunction of the bodies of its derivation rules:

`subjInCurriculum(sc,s,c)` $\rightarrow$ `mandatory(sc)` $\vee$ `optional(sc)`

Unfolding must be recursively applied until all predicates in the *ded* are base. Since we only consider non-recursive derived literals, these unfoldings are guaranteed to always terminate.

After unfolding, we may end up with some negated literal appearing in the RHS as shown by the next example:

$$\phi(x) \rightarrow p(x) \tag{1}$$
$$p(x) \leftarrow q(x) \wedge \neg r(x) \tag{2}$$

When the RHS of the formula 1 is unfolded, it results into:

$$\phi(x) \rightarrow q(x) \wedge \neg r(x) \tag{3}$$

As it can be seen, the formula above is not a proper *ded* since *deds* only contain positive literals by definition. Then, an additional predicate and an additional *ded* are required to fix this situation. More precisely, we must replace the negative literal with a new positive one, and add an extra *ded* as follows:

$$\phi(x) \rightarrow q(x) \wedge forbidR(x) \tag{4}$$
$$forbidR(x) \wedge r(x) \rightarrow \bot \tag{5}$$

When violated, *ded* 4 must be repaired by means of instantiating $q(x)$ and $forbidR(x)$. $forbidR(x)$ can be trivially satisfied by assuming a new ground atom of $forbidR$. This new instance will never cause the violation of any other *ded* since $forbidR(x)$ does not appear in any other *ded*, except 5. According to this formula, *ded* 5 will be irreparably violated if both $forbidR(x)$ and $r(x)$ are satisfied. Therefore, $r(x)$ must never be satisfied when 4 is repaired by means of $q(x)$, which is precisely the semantics of the original formula in 3.

4.1.3. Normalizing deds

For our purposes, we need to guarantee that all the terms of a given atom in a *ded* are variables with different names.

That is, whenever we have a constant in an atom (e.g. *subject(Algebra)*), we replace it by a variable and add a built-in literal binding the variable to the constant (e.g. *subject(x)* $\wedge$ $x = Algebra$). Similarly, whenever we find a variable twice in an atom (e.g. *requires(x,x)*), we replace the second variable for a new one and add some built-in literal to bind them together (e.g. *requires(x,y)* $\wedge$ $x = y$).

We show in Figure 4 all the *deds* required to encode all constraints in our running UML/OCL schema.

4.2. Building the CR-graph from the Deds

Given the *deds* drawn from the constraints in the UML/OCL schema, our intention now is to depict a graph to represent the interactions between the repairs that might be applied to keep a certain constraint (i.e., *ded*) satisfied, and the violations of the constraints that may raise because of that repair. We start by bringing the intuition on when such interactions occur, and then, we show how to encode such interactions in the graph.

Intuitively, the LHS of each *ded* represents the condition for *violating* the constraint, and each conjunction in its RHS represents a different way to *repair* such violation. Thus, there is a repair/violation interaction when some atom from the RHS of a *ded* might unify with a LHS atom of some other *ded*, since by including new instances to satisfy the RHS of the first *ded*, we might accidentally satisfy the LHS of the second *ded*.

To represent this graphically, we build a graph we call CR-Graph. A CR-Graph contains two kinds of vertices: *constraint* vertices, and *repair* vertices. For each *ded*, we build a *constraint vertex* (drawn as a circle) representing its LHS. Then, for each conjunction in the RHS of the same ded, we add one *repair vertex* (drawn as a square) to represents its repairs. Then, we add a discontinuous edge between a repair and a constraint vertex to specify that the repair of the former *ded* might violate the condition stated in the LHS of the latter *ded*.

For instance, in Figure 5 we depict the interaction between the *deds* 23 and 19 of our example. Since repairing the *ded* 23 by inserting an instance of *SubjectInCurriculum* may cause a violation of the *ded* 19, we add a discontinous edge between the first repair vertex to the second constraint vertex. Moreover, to recall that the atom that might cause this violation is *SubjectInCurriculum*, we annotate that edge with the RHS and LHS atoms of *SubjectInCurriculum* that might unify.

Since the CR-graph can be regarded as a visual representation of constraints written as *deds*, we naturally extend the notions/definitions of constraints and schemas to constraint vertices and graphs. That is, given an instantiation I, we say that a constraint vertex is satisfied/violated if its corresponding *ded* is satisfied/violated on I; and we say that a CR-graph is *satisfiable*, if there exists an instantiation I such that satisfies all its constraint vertices.

This graph is based on the *denial constraints dependency graph* from [10], but customized for our purposes. In fact, in the CR-graph we make explicit the different repairs of a constraint. In this way, and differently from [10], we can easily identify those constraints having some repair that does not cause the violation of any other constraint.

In the following we formally define the structure of a CR-graph, we show how to formally represent the *deds* in it, how to represent their interactions, and we provide an algorithm to build it given a set of *deds*.

4.2.1. CR-graph *Structure*

The *CR-graph* is a directed bipartite graph defined by the structure: $<C, R, E_{CR}, E_{RC}, a, f>$, where:

- C is a set of vertices called *constraint vertices*.
- R is a set of vertices called *repair vertices*.
- E_{CR} is a set of directed edges from C to R. Each $c \in C$ may have none, one or several outgoing edges to vertices in R, but each $r \in R$ has exactly one ingoing edge. For any $e \in E_{CR}$, we refer to the source of the edge by $s(e)$, and the target by $t(e)$. Additionally, we define *repairs(c)* $= \{r \in R \mid \exists e \in E_{CR}, s(e) = c \wedge t(e) = r\}$.

```
% Attributes Minimum Cardinality 1
1) curriculum(c) → curriculumName(c, n)
2) curriculum(c) → curriculumYear(c, y)
3) subject(s) → subjectName(s, n)

% Attributes Maximum Cardinality 1
4) curriculumName(c, n1) ∧ curriculumName(c, n2) ∧ n1<>n2 → ⊥
5) curriculumYear(c, y1) ∧ curriculumYear(c, y2) ∧ y1<>y2 → ⊥
6) subjectName(s, n1) ∧ subjectName(s, n2) ∧ n1<>n2 → ⊥

% Attributes Integrity Reference
7) curriculumName(c, n) → curriculum(c)
8) curriculumYear(c, y) → curriculum(c)
9) subjectName(s, n) → subject(s)

% Association Min Cardinalities
10) curriculum(c) → subjInCurriculum(sc, s, c)
11) subject(s) → subjInCurriculum(sc, s, c)

% Association Integrity Reference
12) requires(s0, s1) → subject(s0)
13) requires(s0, s1) → subject(s1)
14) subjInCurriculum(sc, s, c) → curriculum(c)
15) subjInCurriculum(sc, s, c) → subject(s)

% Association Class Primary Key
16) subjInCurriculum(sc1,s,c) ∧ subjInCurriculum(sc2,s,c) ∧ sc1<>sc2 → ⊥

% Hierarchy constraints
17) optional(sc) → subjInCurriculum(sc, s, c)
18) mandatory(sc) → subjInCurriculum(sc, s, c)
19) subjInCurriculum(sc, s, c) → optional(sc) ∨ mandatory(sc)
20) optional(sc) ∧ mandatory(sc) → ⊥

% OCL constraints
21) curriculumName(c1, n) ∧ curriculumName(c2, n) ∧ c1<>c2 → ⊥
22) subjectName(s1, n) ∧ subjectName(s2, n) ∧ s1<>s2 → ⊥
23) subjInCurriculum(sc1,s1,c) ∧ requires(s1,s2) → subjInCurriculum(sc2,s2,c)
24) mandatory(sc1) ∧ subjInCurriculum(sc1, s1, c) ∧ requires(s1, s2) ∧
subjInCurriculum(sc2, s2, c) → mandatory(sc2)
```

Figure 4: *Deds* of the constraints of the UML/OCL schema

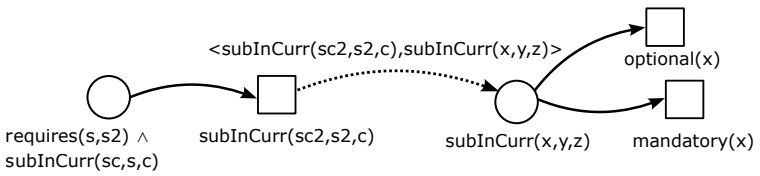


Figure 5: Graph representation of two dependencies interaction

- E_{RC} is a set of directed edges from R to C. Each $r \in R$ may have none, one or several outgoing edges to vertices in C, and each $c \in C$ may have none, one or several ingoing edges. For any $e \in E_{RC}$, we refer to the source of the edge by $s(e)$, and the target by $t(e)$. In addition, we define *violates(r)* = $\{c \in C \mid \exists e \in E_{RC}, s(e) = r \wedge t(e) = c\}$.
- f is a function to label the vertices with a conjunction of literals: $C \cup R \mapsto \Phi$.
- a is a function to label E_{RC} edges with a pair of atoms: $E_{RC} \mapsto Atom \times Atom$. Such function maps each edge $e \in E_{RC}$ to a pair of atoms $<a_r, a_c>$ s.t. $a_r \in f(s(e))$ and $a_c \in f(t(e))$ and a_r and a_c are atoms with the same predicate.

Note that there might be multiple edges between a repair vertex $r \in R$ to the same constraint vertex $c \in C$ since the atoms of one repair vertex might unify with different atoms of the same constraint vertex.

4.2.2. Specifying Deds *in the CR-Graph*

Given a *ded*, we encode its LHS as a constraint vertex $c \in C$, using f to attach to c the conjunction of literals of its LHS. We also encode the (possibly many) conjunctions of its RHS as repair vertices $r \in R$, using f to attach its corresponding conjunction of literals to them.

Then, we link the constraint vertex representing the conjunction of the LHS to the different repair vertices representing the different conjunctions of the RHS of the same *ded* by means of adding edges in E_{CR}. Since each *ded* has one constraint vertex, and each constraint vertex represents the LHS of one *ded*, we can identify a constraint vertex by means of its *ded* and vice versa.

4.2.3. Stating the Interactions in the CR-Graph

We use the edges in E_{RC} to state the repair/violation interactions between *deds*. Given an edge $e \in E_{RC}$, its intended meaning is that when adding instances to satisfy $f(s(e))$, $f(t(e))$ may hold too, and thus, the *ded* of $t(e)$ might be violated.

Intuitively, it is easy to see that an insertion of an instance of a predicate p may only violate those *deds* that have atoms of p in its LHS. Thus, we add an edge e in E_{RC} from every repair vertex r to any constraint vertex c (i.e., $s(e) = r$ and $t(e) = c$) for every pair of atoms $a_r \in f(r)$, $a_c \in f(c)$ s.t. a_c and a_r share the same predicate. To remember this problematic pair of atoms, we add the mapping $a(e) = <a_r, a_c>$in the function a.

4.2.4. Algorithm for Building the CR-graph

The whole process of building the *CR-graph* is defined in Algorithm 1. We show in Figure 6 the graph obtained from the *deds* in Figure 4. For the sake of understandability, we have omitted in the figure multiple edges between the same vertices and the vertices/edges annotations (i.e., functions f and a). Instead, we have attached to each constraint vertex a number to identify the *ded* it is specifying.

As it can be seen, the algorithm creates a CR-graph that grows polynomially with the number of input *deds*. In particular, the algorithm creates D constraint vertices (with D=*number of denials*), $R = D*r$ repair vertices (with r =*average number of disjunctions in the RHS denials*), and at most $E = D * R * l_{LHS} * l_{RHS}$ edges (with l_{LHS}=*maximum number of literals in the LHS* of the deds, and l_{RHS}=*maximum number of literals in the RHS of the deds*). Since the number of *deds* of a UML/OCL schema grows at most polinomially with the size of the schema, it can be easily shown that the final number of CR-graph vertices/edges grows polinomially w.r.t. the original schema.

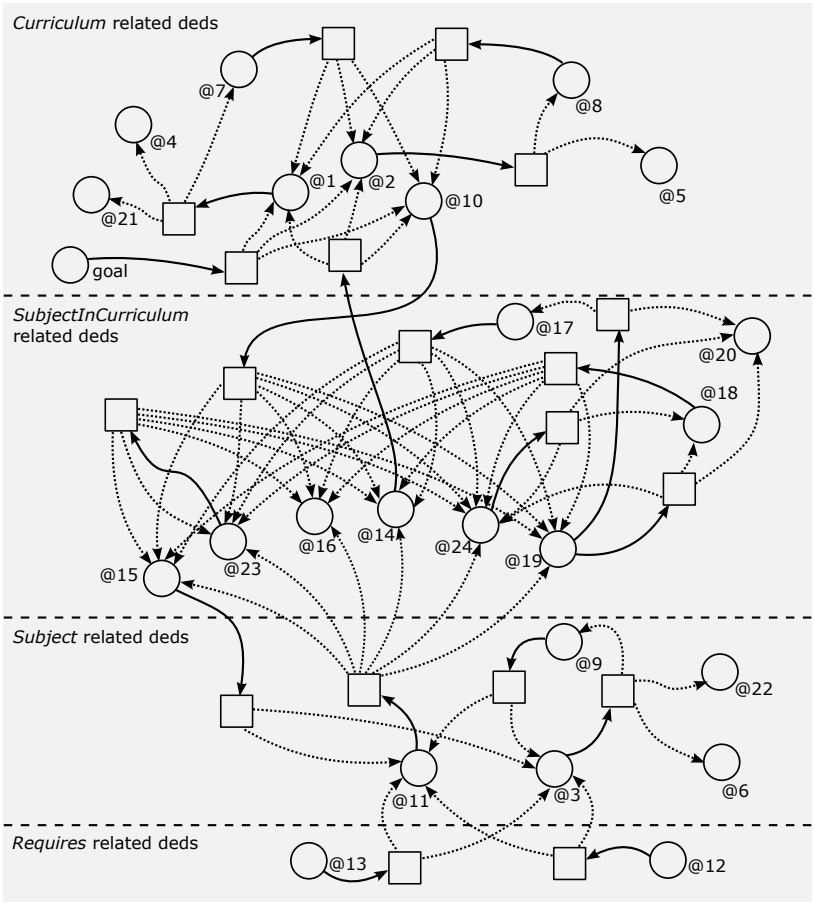


Figure 6: CR-graph of the *deds* schema of the example with *curriculum(x)* as goal

Algorithm 1 getCR-Graph(Set<Ded> D)

CR-graph $G := < C, R, E_{CR}, E_{RC}, f, a >$
//1. Representating *deds* in the graph
for all *Ded* $d \in D$ **do**
  c = new ConstraintVertex()
  C.add(c)
  f.add($c \rightarrow d.LHS$)
  for all Conjunction $\psi \in d.RHS$ **do**
    r = new RepairVertex()
    R.add(r)
    f.add($r \rightarrow \psi$)
    e_{CR} = new Edge(c, r)
    E_{CR}.add(e_{CR})
  end for
end for
//2. Representating *deds* interaction
for all RepairVertex $r \in R$ **do**
  for all ConstraintVertex $c \in C$ **do**
    for all Atom $a_r \in f(r)$ **do**
      for all Atom $a_c \in f(c)$ **do**
        if a_r.predicate = a_c.predicate **then**
          e_{RC} = new Edge(r, c)
          E_{RC}.add(e_{RC})
          a.add($e \rightarrow < a_r, a_c >$)
        end if
      end for
    end for
  end for
end for
return G

5. Simplifying the CR-graph

Our goal now is to remove *deds* without affecting the satisfiability of the goal. Since removing *deds* corresponds to removing constraint vertices from the CR-graph (and its corresponding repair vertices), we directly speak about removing constraint vertices.

To ensure that removing a constraint vertex does not affect the satisfiability of the goal, we need to include the goal in the CR-graph. This is done by considering a new *ded* that encodes the goal in the graph before applying the simplification process. In this manner, we can say that our target is to remove constraint vertices from the graph without affecting its satisfiability.

Then, the main idea is that a constraint vertex c can be removed without affecting the satisfiability of the graph if there is no path from the constraint vertex representing the goal to c, or if c has a repair vertex not pointing to any other constraint vertex. Intuitively, the first case corresponds to deleting a *ded* that might never be violated when looking for an instantiation satisfying the goal, and the latter, corresponds to deleting a *ded* that, when violated, can always be repaired without violating any other *ded*.

According to the previous conditions, the fewer the number of edges, the better will be the removal of constraint vertices. Thus, before starting to remove constraint vertices, the simplification process tries to remove some of the E_{RC} edges of the graph. Indeed, although an edge in $e \in E_{RC}$ is intended to encode the fact that applying the repair encoded in $s(e)$ might cause the violation of the constraint vertex $t(e)$, in some cases it is possible to ensure that such violation never occurs or that can be always avoided.

In Algorithm 2, we summarize the workflow of the simplification process given a CR-graph and a goal to satisfy. Note that the steps of removing edges and removing constraints are applied iteratively until no more deletions can be applied in the graph.

Algorithm 2 getSimplifiedCR-Graph(CR-Graph G, Formula *goalToSatisfy*)

```
G := G.copy()
G.addDeds(getDeds(goalToSatisfy))
while G has some edge or vertex that can be removed do
   G := removeEdges(G)
   G := removeConstraintVertices(G)
end while
return G
```

In the following, we first describe how to add the goal into the CR-graph. Then we explain how we remove edges and vertices. Finally, we show the result of applying these simplifications in our running example. The proofs of correctness of the conditions we apply to remove edges and vertices can be found in the Appendix.

5.1. Adding the Goal to the CR-graph

Recall that a goal specifies a property to be checked about the UML/OCL schema. In terms of logics, a goal is specified as a logic formula *goalToSatisfy()*, for which we want to find an instantiation I satisfying it.

Goals can be formalized in logics by means of a denial constraint stating that there is a violation if the goal is not satisfied, as shown in the following formula:

$$\neg \texttt{goalToSatisfy} \rightarrow \bot$$

Then, they can be translated into *deds* as explained in Section 4.1.2, giving raise to *deds* of the form:

$$\top \rightarrow \texttt{goalToSatisfy}$$

For example, checking whether *Curriculum* is lively give raise to the following *ded*:

$$\top \rightarrow \texttt{curriculum(x)}$$

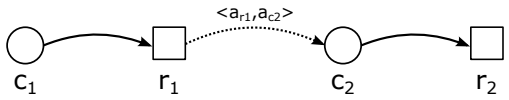


Figure 7: Structure of the vertices for checking the conditions

Note that, in this way, looking for an instantiation I satisfying these *deds* corresponds to look for some instantiation I satisfying the formula *goalToSatisfy()*. Indeed, the LHS of these *deds* are always trivially satisfied, so, any instantiation I will violate these *deds* unless their RHS, that is, the intended goal, hold also in I.

Then, we add in the graph the goal encoded as a *ded*, together with the edges that represents the constraint/repair interactions that might occur with the rest of *deds* from the graph. That is, we add the E_{RC} edges from the goal to the *deds* that might be violated when satisfying the goal.

From now on, we assume that the CR-graph in our running example includes the vertices and edges required to check the goal *Curriculum(x)*, i.e., the goal required to check liveliness of *Curriculum*.

5.2. Removing Irrelevant Edges

The E_{RC} edges in the CR-graph state whether a repair of a constraint may cause the violation of another one. As we have seen, this situation is identified whenever there is a *ded* having an atom in its RHS whose predicate also appears in the LHS of another *ded*.

Our intention now is to remove some of these E_{RC} edges. Indeed, a deeper analysis of the interaction allows to determine that, some repairs r, when applied, does not encompass the violation of their successor constraint vertices $violates(r)$, or that such violations can be avoided. We call these removable edges to be *irrelevant edges*.

For instance, consider the repair vertex of *ded* 1 in Figure 6. Such vertex points to the constraint vertices of *deds* 4 and 21. However, this repair will never violate the former, and can easily avoid the violation of the latter. As it can be seen, *ded* 1 encodes the constraint stating that each *Curriculum* must have at least one *name*, and *ded* 4 encodes that each *Curriculum* should have at most 1 name. Clearly, a maximum cardinality constraint cannot be violated when repairing a minimum cardinality constraint for the same element of the schema. Similarly, *ded* 21 states that curriculums are identified by their *names*. Then, this constraint violation can be easily avoided if we use a new fresh name every time we need to repair 1. Thus, both edges are *irrelevant* regarding the process of satisfiability checking.

In the following, we present several conditions for detecting irrelevant edges. Figure 7 depicts the structure of the (sub)graph in which we apply them. Each condition is intended to remove some edge linking a repair vertex r_1 with a constraint vertex c_2. In addition to the vertices r_1 and c_2 and to the edge $<a_{r1},a_{c2}>$, our conditions depend also on the constraint vertex c_1 parent of r_1 and on some repair vertex r_2 for c_2.

For all these conditions, we assume that θ is the non-ground substitution that makes $a_{r1} = a_{c2}\theta$. Note that θ always exists since a_{r1} and a_{c2} are atoms of the same predicate whose terms are all different variables. Moreover, given a repair vertex r, we denote by $existentials(r)$ the set of all the existential variables that appear in $f(r)$, i.e., those variables appearing in $f(r)$ but not in $f(c)$, where c is the constraint vertex parent of r. We denote by $existentials(a)$, where a is an atom of a repair vertex r, the subset of variables $existentials(r)$ appearing in a.

Condition 1. *The edge $<a_{r1},a_{c2}>$ is irrelevant if there exists a valid substitution θ_2 such that $f(r_2)\theta\theta_2 \subseteq f(c_1) \cup f(r_1)$, where $dom(\theta_2) \subseteq existentials(r_2)$.*

Intuitively, every time $f(c_1)$ holds and the *ded* is repaired by means of instantiating $f(r_1)$, $f(r_2)$ also holds because $f(r_2)$ is subsumed by $f(c_1) \cup f(r_1)$. Thus, despite $f(c_2)$ may be true because of the insertions of $f(r_1)$, c_2 is not violated because $f(r_2)$ is also true.

For instance, whenever we violate *ded* 1 and we repair it by means of *curriculumName(c, n)*, it happens that *curriculum(c)* is true. Therefore, *ded* 7 will never be violated because its repair *curriculum(c)* is already true. Thus, the edge between *deds* 1 and 7 is irrelevant and can be removed.

Condition 2. *The edge $<a_{r1},a_{c2}>$ is irrelevant if for all valid substitution θ'_2 from terms of $f(r_1)$ to terms of $f(c_2)$, there exists another valid substitution θ_2 s.t. $f(r_1)\theta_2 \subseteq f(c_2)\theta \setminus f(r_1)\theta'_2$, where $dom(\theta_2) \subseteq existentials(r_1)$.*

Consider an instantiation I in which $f(c_1)$ holds but $f(r_1)$ does not. So, consider that we instantiate $f(r_1)$ and we add these instances into I to repair c_1. Lets call this new instantiation I'. We now show by contradiction that I' will never violate c_2. Roughly, assume that c_2 is violated in I'. Now, take out from I' the instantiation of $f(r_1)$. Note that I' without this instantiation is precisely I. Then, according to Condition 2, I contains some other instantiation of $f(r_1)$. This means that I does not violate c_1, which is a contradiction.

In our example, *ded* 1 states that each curriculum should have one name, and *ded* 4 states that it should be, at most, one name. Now, consider that I violates *ded* 1 and we repair it by inserting a new instance of *curriculumName(sp, n)*. Lets call this new instantiation I'. Now, we show that I' does not violate *ded* 4 by contradiction. Indeed, if I' violated *ded* 4, this would mean that I' has two instances of *curriculumName(sp, n)*. Consider that we take out the previously inserted instance of *curriculumName(sp, n)*, so, we now have that I has one instance of *curriculumName(sp, n)*, and thus, I does not violate *ded* 1, which is a contradiction. Thus, repairing *ded* 1 cannot violate *ded* 4, and therefore, the edge between *deds* 1 and 4 is irrelevant.

For the third condition, we need to introduce some terminology first. Given a conjunction of (possibly non-ground) literals ϕ, some predicate p and a variable x, the function *instancesUpperBound* (resp. *instancesLowerBound*) returns the maximum (resp. the minimum) number of ground atoms of p that we need to instantiate to make true every literal of predicate p having the variable x in ϕ. As an example, given $\phi = p(x,y) \wedge p(x,z) \wedge p(x,w) \wedge y \neq z$ we have that *instancesUpperBound(ϕ,p,x)* $= 3$ while *instancesLowerBound(ϕ,p,x)* $= 2$.

Condition 3. *The edge $<a_{r1},a_{c2}>$ is irrelevant if there is a variable $x \in existentials(a_{r1})$ whose domain is infinite s.t., for any edge $<a'_{r1},a_{ci}>$ between r_1 and any c_i, where a'_{r1} contains x, there is a predicate p in $f(c_i)$ such that $instancesUpperBound(f(r_1),p,x) < instancesLowerBound(f(c_i)\theta_i,p,x)$, where θ_i is the substitution $a'_r = a_{ci}\theta_i$.*

This condition ensures that the repair obtained from $f(r_1)$ by assuming a new fresh value for x (which always exists because its domain is infinite) will never violate c_2. This is because to violate $f(c_2)$ we need more facts of p with the value x than those created when instantiating $f(r_1)$. Note that as long as x is instantiated with a new fresh value, the unique instances containing x are those created by $f(r_1)$.

In our example, the repair of *ded* 1, *curriculumName(c, n)*, points to *ded* 21 which states that there cannot be two *Curriculum* with the same name. However, *ded* 21 will not be violated after repairing *ded* 1 if we use a new fresh name for n in such a repair. Thus, the edge between *deds* 1 and 21 is irrelevant. Note that this case cannot be detected by condition 2 since c is not an existential variable in *ded* 1.

Condition 4. *The edge $<a_{r1},a_{c2}>$ is irrelevant if there exists a variable $x \in existentials(a_{r1})$ whose domain is infinite s.t. x appears in a built-in-literal of $f(c_2)\theta$ constraining its minimum value (or maximum value), and, x does not appear in any built-in-literal of $f(r_1)$, and for any other c_i pointed to by r_1 by means of an edge $<a'_{r1},a_{ci}>$ such that a'_{r1} contains x, $c_i\theta_i$ contains a built-in-literal constraining its minimum value (or maximum value, respectively), where θ_i is the substitution $a'_{r1} = a_{ci}\theta_i$.*

The previous condition ensures that we can always properly instantiate x in $f(r_1)$ to falsify one of the built-in-literals of $f(c_2)$ (or any other $f(c_i)$ depending on the value chosen for x) if the domain of x is infinite.

For example, consider a new *ded* constraining the *Curriculum* to be, at least, from the year 2000. I.e., *curriculumYear(c, y)* $\wedge y < 2000 \rightarrow \bot$. Then, the repair vertex of *ded* 2 ($curriculumYear(c,y)$) would point to this new *ded*. However, if we pick a value for y greater than 2000, repairing *ded* 2 will avoid the violation of this *ded*. Thus an edge from *ded* 2 to it would be irrelevant.

5.3. Removing Constraint Vertices

We give now three characterizations of constraint vertices that can be removed from the CR-graph without altering its satisfiability. In particular, we formally state each characterization under the form of a

theorem whose proof is given in the Appendix. It is worth noting that removing constraint vertices entails deleting its repair vertices too. This is because we no longer need the repairs of a constraint that has been removed.

To state these characterizations, we need to introduce some terminology. A constraint vertex c is *empty* if it has attached the formula $\top$ (i.e., $f(c) = \top$). As we have seen, there is always one such vertex because the one encoding the goal is always empty.

The intuition behind the first characterization is that a non-empty constraint vertex c can be removed if there is no path from an empty constraint vertex c_0 to it. Indeed, consider the empty instantiation $I_0 = \emptyset$. Such instantiation does not violate c since $f(c) \neq \top$, but violates all the empty vertices c_0 in the CR-graph (that is, the *goal* among others). These constraints will have to be repaired by means of adding new instances in I, which might violate some other constraints that will need to be repaired too. However, the unique constraints that might be violated due to this process are those constraints reachable from c_0. Since c is not reachable from c_0, c is never going to be violated and thus, it is not an impediment to proof the satisfiability of the graph. Formally,

Theorem 2. *Let c be a constraint vertex s.t. $f(c) \neq \top$ and for any constraint vertex c_0, where $f(c_0) = \top$, there is no path from c_0 to c. Then, deleting c preserves the satisfiability of the graph.*

Constraint vertices for *deds* 12 and 13 in the CR-graph of Figure 6 satisfy the previous condition. Therefore, they can be removed without affecting the satisfiability of the CR-graph (i.e., without affecting whether *Curriculum* is lively or not).

The second characterization, formalized in Theorem 3, affects those constraint vertices c containing some predicate p not appearing in any repair vertex. These vertices will never be violated by the empty instantiation $I_0 = \emptyset$ nor by any other instantiation I_i obtained by repairing I_0 since no such I_i will ever contain an instance of p. Therefore, those vertices can also be safely removed. Formally,

Theorem 3. *Let c be a constraint vertex, and p some predicate appearing in $f(c)$. If for all repair vertex $r \notin repairs(c)$, r does not contain the predicate p in $f(r)$, then, deleting c preserves the satisfiability of the graph.*

As an example, the constraint vertex for *ded* 23 in Figure 6 contains the predicate *requires* which does not appear in any of the repair vertices of the CR-graph. Therefore, it can be safely removed.

Finally, assume a constraint vertex c having some repair vertex r not pointing out to any constraint vertex. Then, whenever c is violated, it can be repaired without violating any other constraint. Therefore, c is not an impediment to find an instantiation I satisfying the whole schema and, hence, it can be removed. Formally,

Theorem 4. *Let c be a constraint vertex with some repair vertex $r \in repairs(c)$ s.t. $violates(r) = \emptyset$. Then, deleting c preserves the satisfiability of the graph.*

For instance, after removing all the irrelevant edges, *ded* 1 has a repair vertex not pointing to any other constraint vertex. Thus, this *ded* can be safely removed.

5.4. Applying the Simplifications to our Example

In the following, we discuss the results of our simplification technique using our running example and two different goals: checking whether *Curriculum* is lively, and checking whether *Requires* is lively.

In the first case, we get the empty CR-graph after simplification. This directly means that *Curriculum* is lively in the original UML/OCL schema and, thus, there is no need to apply existing reasoning techniques. Note that this is an important benefit since these reasoning techniques tend to be rather time consuming.

In the second case, we get the non-empty simplified CR-graph we show in Figure 8. The resulting graph is much smaller than the original one since it contains only five constraint vertices. Taking into account that one of this constraint vertices represents the goal, this means that, to analyze whether *Requires* is lively, we only need to consider a schema including the four constraints shown in the CR-Graph. Since this schema is much simpler, the reasoning tools perform faster on it rather than on the original schema.

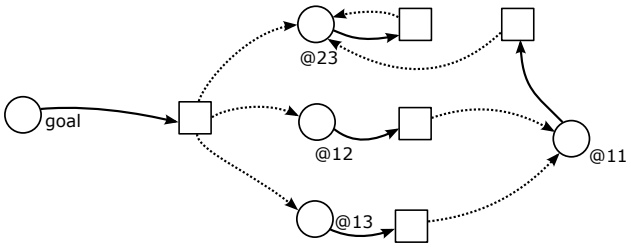


Figure 8: Simplified CR-graph for the goal *requires(s,s2)*

6. Building the Simplified UML/OCL Schema

The main goal of our approach is to obtain a simplified version of the original UML/OCL schema that allows current techniques to check a given property more efficiently than in the original schema while preserving the satisfiability of the check. This satisfiability preservation is guaranteed by the simplifications of the CR-graph we have proposed in the previous section. So, we now have to build the new UML/OCL schema from the simplified CR-graph.

The basic idea to obtain the UML/OCL simplified schema is to build the subset of the original UML/OCL schema containing all the constraints that appears in the simplified CR-Graph. Such process requires a careful treatment to ensure that the new UML/OCL schema is syntactically correct. This is achieved by means of three different steps:

- Including a UML class, attribute or association for each predicate still in the CR-graph.
- Adding to the UML class diagram the *deds* in the CR-graph representing graphical UML constraints.
- Reconstructing the surviving OCL constraints of the graph, considering the new UML class diagram obtained in the previous steps.

6.1. Building the Simplified UML Class Diagram

The idea is to start from an empty UML class diagram, and add a UML element for all the predicates that remains in the simplified CR-graph.

In particular, for each predicate C representing a class C, we include the class C in the new diagram. For each predicate *CAttr* representing an attribute *Attr* for a class C, we add C in the diagram if it is not yet there and we add *Attr* in C. Then, for each predicate *Assoc* representing an association (or association class) *Assoc* between classes $C_1, ..., C_n$ we incorporate in the diagram the classes $C_1, ..., C_n$ if necessary, and we add *Assoc* in the schema as well.

Note that when including an association or an association class to the diagram their implicit key and referential integrity constraints will already be stated by the diagram. Thus, this step also adds the UML implicit constraints that may appear in the simplified CR-graph.

In our example for checking whether *Requires* is lively, according to the simplified CR-graph of Figure 8, the new UML class diagram contains the classes *Curriculum*, and *Subject* (but not *Mandatory*, neither *Optional*), the association class *SubjectInCurriculum*, the association *Requires*, and no attribute. Note additionally that, just by adding the association *Requires*, we have already added in the schema the constraints corresponding to the *deds* 12 and 13 (that is, the UML integrity reference implicit constraints) appearing in our simplified graph.

6.2. Adding UML Graphical Constraints to the Class Diagram

Once we have initialized the class diagram, we can add the graphical UML constraints corresponding to the *deds* that are still in the simplified CR-graph. We distinguish three different kinds of such constraints:

- *Minimum/Maximum Cardinality Constraints:* if a *ded* representing a cardinality constraint appears in the simplified CR-graph, then, its association/attribute predicate also appears in the CR-Graph, and thus, the association/attribute appears in the previously created class diagram. Then, we can naturally add this cardinality constraint to this element of the diagram.

- *Hierarchy Constraint:* similarly, if a *ded* representing a hierarchy constraint appears in the simplified CR-graph, then, both its subclass and superclass appears in the initialized class diagram. So, we can directly include the hierarchy in the diagram.

- *Disjointness/Completeness Constraint:* like in the previous case, if a *ded* representing a disjoint/complete constraint appears in the simplified CR-graph, then, the subclasses and the superclass involved appears in the created class diagram. However, the hierarchy relation between the subclasses and superclass might not be present. Thus, we first add the hierarchy relation between the involved classes if it is not yet in the diagram, and then, we add the corresponding disjoint/complete constraint.

In our example, for checking the liveliness of the *Requires* association, we had to add a minimum cardinality 1 for the *Curriculum* member end of *SubjectInCurriculum* association because of the surviving *ded* 11.

6.3. Reconstructing the OCL Constraints

In the last step, we incorporate to the UML schema all the OCL constraints whose *ded* is still in the CR-graph. However, the OCL expression specifying the constraint might need to be rewritten since the simplified UML class diagram is different to the original one and, thus, the original OCL constraint might not be syntactically correct according to the new schema. Briefly, there are two types of syntactic problems that might happen: type conformance problems, and cardinality conformance problems.

6.3.1. Type conformance problem

An OCL expression may contain an object/set reference whose type does not conform to the expected type. E.g. consider an OCL expression such as `e.manager = boss`, where `e.manager` has the type `Employee` and `boss` has the type `Boss`. `Boss` might conform to `Employee` in the original schema because of a hierarchy between the two. However, the hierarchy may have been removed in the simplification process, thus, making this OCL constraint syntactically incorrect due to a type conformance problem. To solve this situation, we can include in the UML class diagram the necessary original hierarchy constraints.

6.3.2. Cardinality conformance problem

An OCL expression that was initially retrieving a single value may return, after the simplification of the schema, a set of values. E.g. given the expression `e.manager.oclIsKindOf(Boss)`, it might happen that `e.manager` was a single object in the original schema (because of a maximum cardinality constraint), but that it is now a collection (because the max. cardinality constraint has been removed). Thus, this OCL expression is no longer retrieving a single boolean value, but a bag of boolean values. This may lead to a syntactic problem if such expression is used in a place where a single boolean expression is expected. Like the previous case, we can solve these kind of problems by adding the necessary maximum cardinality constraints. However, we propose taking a different strategy in order to keep as much as possible our simplifications.

An object reference that has turned into a collection reference may be placed in a *source* of some OCL operation, or in some operation *argument*. We treat each case differently.

If the object reference appeared in the source, we simply iterate the source with a `forAll` and apply the operation to each object of the set. Thus, in our example, we get the following OCL expression: `e.manager->forAll(m|m.oclIsKindOf(Boss))`, which now returns a single boolean value. This rewriting does not alter the semantics of the constraint since both expressions give raise to the same logical *ded*. If the operation is not boolean but a cast (i.e. `oclAsType`), we will just change the expression `oclAsType` for `selectByKind`. Again, this modification does not alter its logic *ded* translation, but permits fitting to the syntax required by the simplified schema.

If the object reference appeared in an operation argument, the unique possibilities are that the operation is an OCL `includes` or an OCL `excludes`, since they are the unique OCL operations using a single object as argument. In the case of `excludes`, we replace it by `excludesAll`, which preserves the logic translation to a *ded*. To deal with `includes`, we replace it by `includesAll`. However, in this last case we have to add an extra condition to force the argument not to be empty (otherwise both expression would evaluate differently in OCL). Thus, for instance, the expression `self.employee->includes(self.boss)` is translated as `self.employee->includesAll(self.boss)` and `self.boss->notEmpty()`. This modification also preserves the semantics because the translation to *deds* remains the same.

7. Experiments

To show the feasibility of our approach and the efficiency improvement it provides, we have implemented our method in a prototype tool. Then, we have used this tool to reason about two different UML/OCL schemas:

- The DBLP schema [15]: a UML/OCL schema with 30 classes/associations specifying the information system of the well-known DBLP computer science bibliography site.
- The EU-Rent schema [16]: a UML/OCL schema with 73 classes/associations specifying the information system of a fictional car rental system.

Given these schemas, we have run two different experiments. The first one consisted in running several liveliness tests on both schemas, and the second, on running several *state rechability tests*. Both experiments were designed in such a way that all tests were of increasing complexity. Moreover, to show that our method is not tied to a concrete reasoning tool, we have run these experiments using two different analyzers: USE [17], and the satisfiability checker of AuRUS [21]: SVTe [25]. Such tools were chosen because they provide a textual API that allowed us to automate the experiments' execution, thus minimizing experiment errors due to human manual intervention.

In summary, we have run more than 500 different tests combining schemas, test goals and reasoning engines. All these tests have been run on an Intel Core i7-4710HQ, with 8GB of RAM, in a Windows 8 operating system.

In the following, we first describe both experiments (design and results) separately, and then, we discuss the results together to draw some conclusions. Afterwards, we point some possible threats to validity.

7.1. Class Liveliness Experiment

We have randomly selected 4 classes for checking their liveliness property. For each class C, we have tested whether having n instances of C was satisfiable. We started the tests with $n = 1$, and we iteratively increased its value until $n = 20$, or until the test execution time exceeded a time threshold of 1 hour. Each n and class C determined a different *goalToSatisfy()*. Hence, for each *goalToSatisfy* we have proceeded as follows:

1. Analyzed the satisfiability of *goalToSatisfy* through USE
2. Analyzed the satisfiability of *goalToSatisfy* through AuRUS
3. Simplified the schema with that particular *goalToSatisfy*
4. Analyzed the satisfiability of *goalToSatisfy* through USE in the simplified schema
5. Analyzed the satisfiability of *goalToSatisfy* through AuRUS in the simplified schema

In the following we discuss the results obtained when performing this experiment in each schema separately.

7.1.1. DBLP Schema

We randomly selected the classes *Authored Book*, *Edited Book*, *Journal Section*, and *Journal Volume* to perform this experiment. In Figure 9 we show the execution times obtained for testing their liveliness, depending on the reasoning engine used. The results obtained using the original schema are depicted with circled dots, while those of the simplified one are in cross dots.

All the results belonging to the simplified schema have better execution times than those belonging to the original schema in both tools. Moreover, reasoning in USE with the simplified schema never reached the threshold of 1 hour while it reached it some times when reasoning with the original schema. AuRUS did not reach the threshold in half of the tests done with the simplified schema.

In Table 1 we show the number of deds of the original and simplified CR-Graph, the elements of the UML/OCL schema that could be removed, the average time taken for simplifying the schema for each *goalToSatisfy* (grouped by the selected class), and the performance improvement. We would like to stress that the number of removed deds remained constant for each selected class despite increasing the number of instances forming the *goalToSatisfy*; moreover, such number was almost constant through the different selected classes (the unique exception is *Journal Section*, for which the number of remaining deds was greater by +2). We also provide the exact total execution time required by each of the tools and the improvement achieved after simplification. Note that this improvement ranges from doubling the speed, to running x369.90 faster. Note also that the simplification time is almost negligible since its average takes less than 0.5s.

Table 1: DBLP - Summary of Execution Time Improvement

	Authored Book	Edited Book	Journal Section	Journal Volume
Orig. CR-Graph *deds*	192	192	192	192
Simp. CR-Graph *deds*	55	55	57	55
Removed classes	2	2	2	2
Removed associations	2	2	2	2
Removed attributes	24	24	24	24
Removed OCL constraints	20	20	19	20
Avg. simp. time	365 ms	464 ms	389 ms	325 ms
USE original time	2,091,884 ms	6,170,988 ms	3,907,566 ms	1,183,988 ms
USE simplified time	528,420 ms	72,447 ms	565,907 ms	548,625 ms
USE gain	x3.96	x85.18	x6.90	x2.16
AuRUS original time	1,124,157 ms	471,045 ms	200,300 ms	863,354 ms
AuRUS simplified time	94,844 ms	66,520 ms	13,933 ms	2,334 ms
AuRUS gain	x11.85	x7.08	x14.38	x369.90

7.1.2. EU-Rent

We randomly selected the classes *Blacklisted*, *Canceled Customer Liable*, *Discount*, and *Rental Duration* to check their liveliness. Figure 10 shows the execution times it took to check them in each reasoning engine. Again, the results obtained in the original schema are depicted with circled dots while those in the simplified one are in cross dots.

In this case, USE was not able to handle the tests in the original schema[2], thus, we cannot provide the time comparison with the simplified schema. However, it is worth noting that USE was able to reason with the simplified schema until test goals consisting of 11 instances.

Regarding AuRUS, note that it runs faster for all tests in the simplified schema, specially in those tests involving *Blacklisted*, *Discount* and *Rental Duration*, for which their execution times took few seconds.

Similarly as before, in Table 2 we show the number of *deds* in the original and in the simplified CR-graph, the UML/OCL elements that could be removed, the average time taken to simplify the schema, and the efficiency improvement for AuRUS. It is worth saying that, again, the number of removed constraints remained constant for each selected class, despite increasing the number of instances forming the *goalToSatisfy*, and

[2]USE requires some manually given instance boundaries in which to perform the search. Using small boundaries, USE determined the goals to be unsatisfiable, while they are satisfiable. Using bigger boundaries, USE run out of memory.

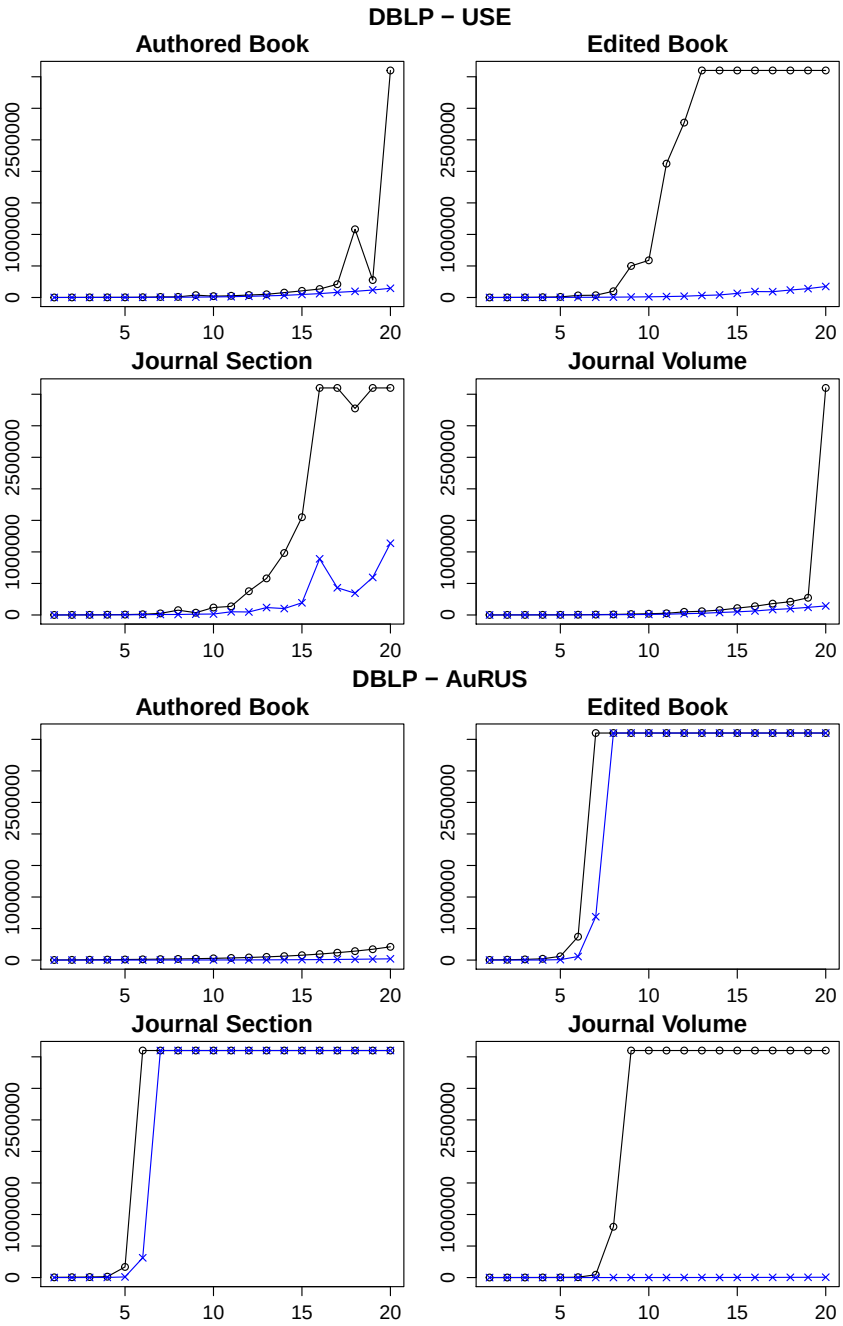


Figure 9: Execution times in ms (y-axis) of the DBLP class liveliness tests per #instances (x-axis)

even between the different classes selected. As it can be seen, the benefit of the simplification ranged from x4.32 to x74.91 faster execution times.

Table 2: EU-Rent - Summary of Execution Time Improvement

	Blacklisted	C. Customer	Discount	Rent. Duration
Orig. CR-Graph *deds*	328	328	328	328
Simp. CR-Graph *deds*	88	88	87	87
Removed classes	15	15	15	15
Removed associations	6	6	6	6
Removed attributes	48	48	48	48
Removed OCL constraints	16	16	16	16
Avg. simp. time	510 ms	469 ms	570 ms	510 ms
AuRUS original time	2,750,198 ms	3,377,253 ms	1,291,168 ms	571,408 ms
AuRUS simplified time	72,357 ms	780,893 ms	17,237 ms	16,820 ms
AuRUS gain	x38.01	x4.32	x74.91	x33.97

7.2. State Reachability Experiment

This experiment started with randomly generating n instances of different elements of the schema. Then, we measured the execution time for testing the satisfiability of a goal containing these n instances. We started with $n = 1$ and we iteratively increased its value until $n = 20$ or until the reasoning tool exceeded a time threshold of 1 hour. To ensure that every test was harder than the previous one, we built the instantiation corresponding to $n = i$ by adding one random instance to that of $n = i - 1$.

Figure 11 shows the execution times taken to perform the state reachability tests in both schemas, according to each reasoning engine. Circled dots stand for results in the original schema while cross dots in the simplified one.

Almost all tests applied in the simplified version of the schema have execution times of few seconds, except when applying USE to check state reachability in EU-Rent, where it reached the execution time threshold when considering 12 instances. Note, however, that in the original schema, USE already reached the threshold with 6 instances.

We show in Table 3 the execution time improvement for each schema and each reasoning tool, together with the size of the simplified CR-Graph when $n = 1$, $n = 5$, $n = 10$, $n = 15$, and $n = 20$. As it can be seen, the simplification process returned the smallest CR-Graph when dealing with $n = 1$ (i.e., the smallest state reachability test), and the size of the CR-Graph slightly increased when increasing the size of the state reachability test. With regarding the benefits of the simplification, the AuRUS tool improved its performance between x13.51 and x18.68 faster, while the USE tool between x4.12 and x6.30.

Table 3: State Reachability - Summary of Execution Time Improvement

	DBLP	EU-Rent
Orig. CR-Graph *deds*	192	328
Simpl. CR-Graph *deds* (n = 1)	55	87
Simpl. CR-Graph *deds* (n = 5)	55	88
Simpl. CR-Graph *deds* (n = 10)	57	92
Simpl. CR-Graph *deds* (n = 15)	57	95
Simpl. CR-Graph *deds* (n = 20)	60	96
Avg. Simp. Time	557,7 ms	582,75 ms
USE Original Time	2,469,801 ms	310,852 ms
USE Simplified Time	600,189 ms	49,320 ms
USE gain	x4.12	x6,30
AuRUS Original Time	217,340 ms	414,721 ms
AuRUS Simplified Time	16,086 ms	22,204 ms
AuRUS gain	x13.51	x18,68

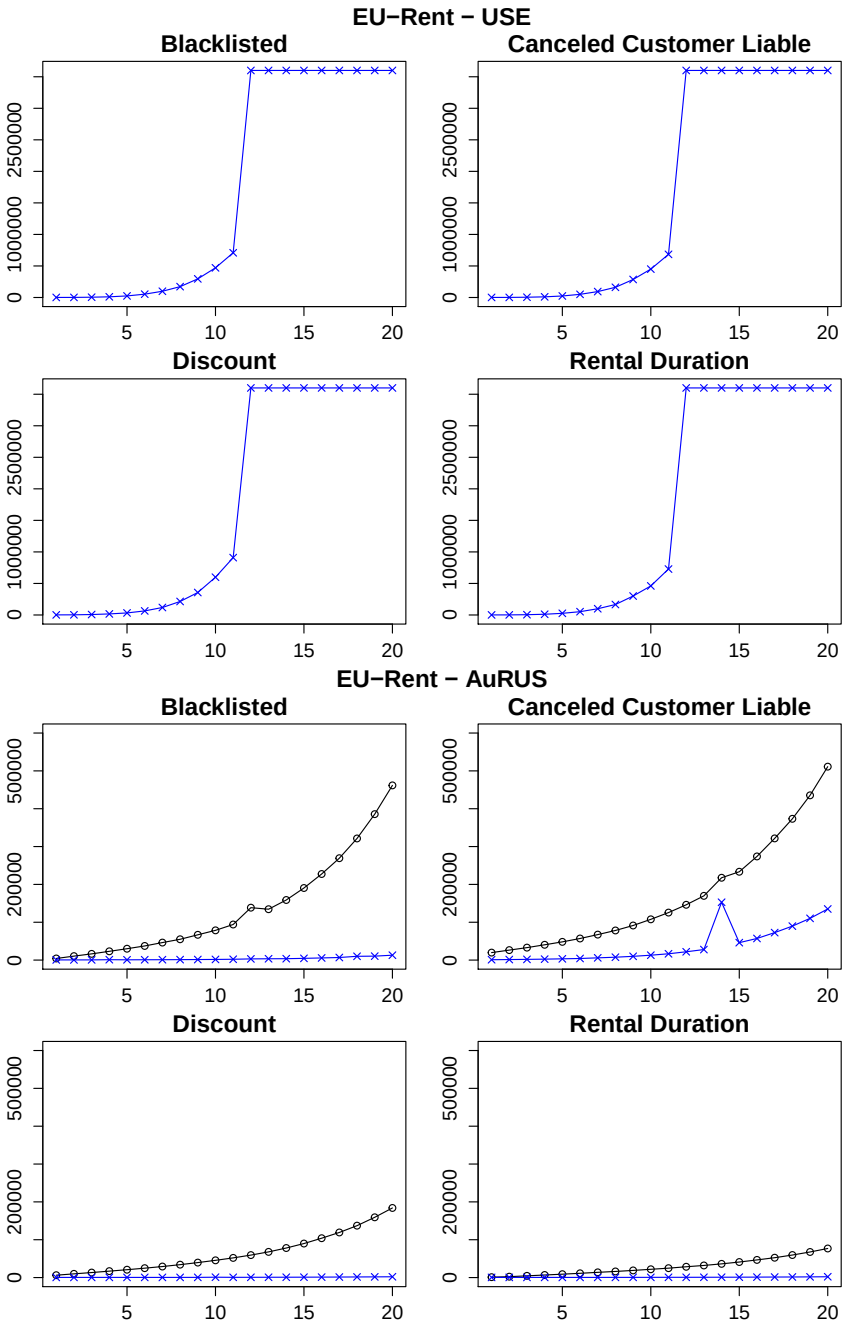


Figure 10: Execution times in ms (y-axis) of the EU-Rent class liveliness tests per #instances (x-axis)

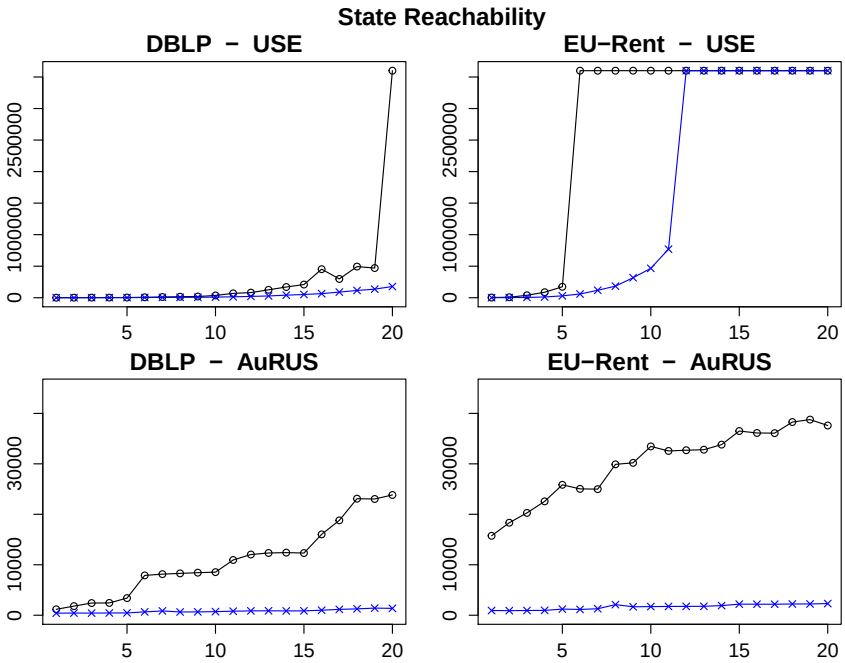


Figure 11: Execution times in ms (x-axis) for the state reachability tests per #instances (y-axis)

7.3. Discussion of the Results

As we have seen, the simplification reduced the execution times in all test cases. That is, the simplification took almost negligible time (less than 0.5s on average) and reported benefits that ranged between x2 and x370 (depending on the particular test and reasoning engine used), not to say that some tests that where not affordable within a time threshold of 1 hour could be done in their simplified versions.

It is of special importance to realize that the simplification enhanced the performances of two different reasoning tools of different kind. Indeed, whereas USE is a tool whose internal reasoner (Kodkod), is based on searching for a consistent instantiation among some instance bounds for each class/association, the AuRUS internal reasoner (SVTe) is based on dynamically maintaining the constraints. Thus, we expect improvements on tools such USE when removing classes/associations, since then we are removing its search space, but we expect improvements on tools such as SVTe when reducing the number of constraints, since then the number of constraints to maintain decreases. However, as we have seen, since the simplification method is able to remove classes/associations and constraints, the simplification allowed improving the time performances of both kinds of tools.

Interestingly, it seems that once the schema is fixed, the CR-graph returned by the simplification process barely varies in function of the selected *goalToSatisfy*. We claim that this phenomenon is due to the fact that, at the end, the *goalToSatisfy* is no more than an extra *ded* of the CR-graph, and all *deds* are treated equally. Thus, the bigger the CR-graph (i.e., the number of the overall *deds*), the lower is the influence of the specific *goalToSatisfy*. In any case, this does not mean that we might be able to simplify without the *goalToSatisfy* (it is easy to see from the theorems that we need the complete CR-graph before start simplifying soundly), but these results tell us that a possible fruitful further work could be defining simplifying theorems that focuses on the *goalToSatisfy*. Intuitively, the simplified schema for testing the liveliness of a class that does not play a key role in a UML/OCL schema (such as *Country* in the EU-Car UML/OCL schema) should be much simpler than the simplified schema for testing the liveliness of a key element of the class diagram (such as *Rental*, in the EU-Car UML/OCL schema).

7.4. Threats to Validity

The major issue that concerned us in the development of such experiments was the usage of academic UML/OCL schemas instead of real industrial ones. In order to mitigate such issue, we have used the DBLP schema (which was obtained through reverse engineering on a real functional system), and then, the fictional EU-Care Rental schema which is a large academic example simulating an industrial situation. In any case, real industrial-size schemas would be required to confirm the results shown previously.

In addition, due to difficulties on configuring a compatible environment where to run the reasoners, the experiments could not be carried in a sandbox environment. That is, the full dedication of the machine to run the tests was not guaranteed, which explains some small peaks that can be observed in the result Figures.

8. Related Work

We found similar approaches to ours in the literature regarding the notion of model *slicing* and also in proposals aimed at syntactically detecting relevant elements for testing some of the properties. There are also other techniques coming from the SAT-problem community that, if implemented, could potentially enhance the UML/OCL reasoning tools. We review all of them in this Section.

8.1. Model Slicing and Simplification

Shaikh et al. [18] proposed a method aimed at *slicing* UML/OCL schemas. That is, given a UML/OCL schema, it returns a set of disjoint UML/OCL schema slices (i.e., subsets of the original schema) so that it is guaranteed that the original schema is satisfiable (i.e., it admits at least a non-empty instantiation) if at least one of its slices is also satisfiable. Therefore, the efficiency improvement of the method relies on the number of slices built.

The main drawback of this method is that one single constraint might make slicing impossible. For instance, assume a constraint saying that the class C has at least one instance (as it happens with *singleton* classes). Then, such class C should appear in every slice. This is because if C was unlively, the whole schema would be unsatisfiable, and thus, each slice should be unsatisfiable too. However, C cannot appear in two different slices since slices are disjoint, thus, the unique possibility is to have one unique slice. To address this issue, an extension of this work sketches a solution with non-disjoint slices[19]. However, this extension is not formally defined nor proved.

Similarly, Sun et al. [20] proposed another slicing technique but they do not formally define which kind of semantics is preserved with the slicing procedure. I.e., they do not state which kind of reasoning (schema satisfiability, class liveliness, etc) is preserved with their method.

The work of Lano et al. [26], takes a UML/OCL schema with operations and a set V of selected attributes and classes of the schema and it returns a unique UML/OCL schema slice that is smaller than the original one. The basic idea is that any sequence of operation executions brings the same values into V when applied in the original and in the sliced schema. Unfortunately, neither a proof of correctness nor an empirical evaluation of the efficiency improvement are provided.

The method proposed by Seiter et al. [27] detects which are the relevant elements to take in account to perform a concrete reasoning task. However, this method does not take into account the semantics of the constraints, but only checks for the attributes/classes/associations accessed by them.

Therefore, as far as we know, none of the current proposals provides a formal proof of the correctness of the slices, as we do for our simplified schema. We understand this is an important drawback because we cannot integrate any simplification/slicing technique into a formal verification/validation tool unless we have a strong mathematical basis ensuring that the output of reasoning has not been altered.

It is worth saying also that [18, 19, 20] incorporate some kind of simplification before slicing. I.e., they first remove irrelevant UML/OCL elements so that they can break the schema into smaller slices. However, this simplification is only intended to deal with frequent patterns (e.g. primary key attributes that can be removed), and, again, there is no formal proof of this simplification. The approach we have proposed in this paper might be adopted by these techniques as a previous step to their slicing procedures.

8.2. Other Techniques

The literature of SAT-reasoning contains plenty of optimization techniques that, if implemented, could have a deep impact in the efficiency of UML/OCL reasoning tools. This is because, at the end, most of these tools relies on some kind of SAT reasoner in its core. For instance, the work by Clarisó et al. [28] showed that, by means of analyzing and straightening the number of possible instances every class/association could have (a technique coming from the Constraint Satisfaction Problem literature), they could enhance the response time of a particular UML/OCL reasoning tool by x50. We think that this technique and ours are complementary since we can first, simplify the UML/OCL schema, and then, perform the previous analysis and straightening in the simplified version of it. However, the approach by Clarisó is strongly tied to CSP reasoners, or any other reasoner requiring to fix the number of instances per class/association to take into account while reasoning. In contrast, our approach is absolutely independent from the reasoning tool since we are not tied to a specific strategy.

In a similar manner, Balaban et al. [29] propose a method to tighten the cardinality constraints. For instance, they tight a maximum cardinality constraint to a lower value when they identify that the maximum value can never be reached. Since this approach directly works with the UML/OCL schema, it is not tied to any particular reasoning tool. However, no evidence of their benefits for reasoning are given since their unique aim was to improve schema readability by writing their *real* cardinalities.

Finally, it is worth to mention that Wahler et al. [30] propose an approach that, starting from an empty UML/OCL schema, allows adding new constraints into it while ensuring that it is satisfiable. Our work, however, goes in the opposite direction: we remove elements from the original schema while preserving its satisfiability (or the satisfiability of some goal). In this way, when our method cannot continue simplifying a UML/OCL schema, it returns a small UML/OCL schema to be reasoned with heavy verification/validation tools; on the contrary, when the method in [30] cannot continue adding new constraints into the schema ensuring its satisfiability, the user has to use heavy verification/validation tools with the schema.

9. Conclusions and Further Work

We have proposed an approach aimed at simplifying a UML/OCL schema for efficient reasoning. That is, given a UML/OCL schema, and some property to test, our method returns a subset of the original UML/OCL schema that preserves the satisfiability of the property.

Our method is based on translating the UML/OCL schema into a *deds* logic representation, and then, building the CR-graph. The CR-graph is a graph whose vertices represents the constraints with its different repairs, and whose edges represents the constraints that might be violated when applying some concrete repair. By means of analyzing the CR-Graph, we remove those constraints that do not compromise the satisfiability of the intended goal. After that, we rebuild the UML/OCL schema that contains those constraints. All these steps have been formally defined and proofed for its correctness.

To show the feasibility of our approach and to illustrate the efficiency improvement it provides, we have developed a prototype tool to automatically obtain the simplified UML/OCL schema. With this tool, we have made some experiments combining different UML/OCL schemas, test goals, and reasoning tools. We have shown that in all the situations, the simplification process improved the time performance for reasoning ranging from x2 to x370 times faster.

There are several directions to extend this work. Briefly, the simplification process can be extended by considering new conditions to remove edges or vertices in the CR-graph. In addition, the CR-graph could be studied as a way to guide current reasoning tools based on building consistent instantiations. Indeed, the CR-graph indicates which repairs induces less constraint violations than others, thus, it could be used to guide reasoners to prefer those repairs. Finally, it would be worth to study which are the kind of constraints that reduces the reasoning times the most to focus particularly on them.

Acknowledgements This work has been partially supported by the Ministerio de Economía y Competitividad, under project TIN2014-52938-C2-2-R and by the Secreteria d'Universitats i Recerca de la Generalitat de Catalunya under 2014 SGR 1534 and a FI grant.

References

[1] Olivé, A.: Conceptual Modeling of Information Systems. Springer, Berlin (2007)
[2] Dobing, B., Parsons, J.: How UML is used. Commun. ACM **49**(5) (May 2006) 109–113
[3] Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley object technology series. Addison-Wesley (2004)
[4] Object Management Group (OMG): Unified Modeling Language (UML) Superstructure Specification, version 2.4.1. (2011) `http://www.omg.org/spec/UML/`.
[5] Object Management Group (OMG): Object Constraint Language (UML), version 2.4. (2014) `http://www.omg.org/spec/OCL/`.
[6] Endres, A., Rombach, H.D.: A handbook of software and systems engineering: empirical observations, laws and theories. Pearson Education (2003)
[7] Hilken, F., Niemann, P., Gogolla, M., Wille, R.: Towards a catalog of structural and behavioral verification tasks for UML/OCL models. In: Modellierung 2016, 2.-4. März 2016, Karlsruhe. (2016) 117–124
[8] Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. Journal of Systems and Software **93**(0) (2014) 1 – 23
[9] Hamann, L., Hofrichter, O., Gogolla, M.: On integrating structure and behavior modeling with OCL. In: International Conference on Model Driven Engineering Languages & Systems (MODELS 2012), Springer (2012) 235–251
[10] Queralt, A., Teniente, E.: Verification and validation of UML conceptual schemas with OCL constraints. ACM Transactions on Software Engineering and Methodology **21**(2) (2012) 13
[11] Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software & Systems Modeling **9**(1) (2010) 69–86
[12] Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artificial Intelligence **168**(1-2) (2005) 70–118
[13] González Pérez, C.A., Cabot, J.: Formal Verification of Static Software Models in MDE: A Systematic Review. Information and Software Technology (March 2014)
[14] Lange, C., Chaudron, M.R.V., Muskens, J.: In practice: UML software architecture and design description. Software, IEEE **23**(2) (March 2006) 40–46
[15] Planas, E., Olivé, A.: The DBLP case study (2006) `http://guifre.lsi.upc.edu//DBLP.pdf`.
[16] Frías, L., Queralt, A., Olivé, A.: EU-rent car rentals specification. Technical report, Universitat Politècnica de Catalunya (12 2003)
[17] Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating SAT solving into use. In: Objects, Models, Components, Patterns. Springer (2011) 290–306
[18] Shaikh, A., Clarisó, R., Wiil, U.K., Memon, N.: Verification-driven slicing of UML/OCL models. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. (2010) 185–194
[19] Shaikh, A., Wiil, U.K., Memon, N.: UOST: UML/OCL aggressive slicing technique for efficient verification of models. In: System Analysis and Modeling: About Models. Springer (2011) 173–192
[20] Sun, W., France, R., Ray, I.: Contract-aware slicing of UML class models. In: Model-Driven Engineering Languages and Systems. Volume 8107 of Lecture Notes in Computer Science. Springer (2013) 724–739
[21] Rull, G., Farré, C., Queralt, A., Teniente, E., Urpí, T.: Aurus: explaining the validation of UML/OCL conceptual schemas. Software and System Modeling **14**(2) (2015) 953–980
[22] Andrews, P.: An Introduction to Mathematical Logic and Type Theory. Applied Logic Series. Springer (2002)
[23] Mecca, G., Rull, G., Santoro, D., Teniente, E.: Ontology-based mappings. Data & Knowledge Engineering **98** (2015) 8–29
[24] Franconi, E., Mosca, A., Oriol, X., Rull, G., Teniente, E.: Logic foundations of the OCL modelling language. In: Logics in Artificial Intelligence. Volume 8761 of Lecture Notes in Computer Science. Springer (2014) 657–664
[25] Farré, C., Rull, G., Teniente, E., Urpí, T.: SVTe: a tool to validate database schemas giving explanations. In: 1st International Workshop on Testing database systems. DBTest '08, New York, NY, USA, ACM (2008) 9:1–9:6
[26] Lano, K., Kolahdouz-Rahimi, S.: Slicing of UML models using model transformations. In: Model Driven Engineering Languages and Systems. Volume 6395 of Lecture Notes in Computer Science. Springer (2010) 228–242
[27] Seiter, J., Wille, R., Soeken, M., Drechsler, R.: Determining relevant model elements for the verification of UML/OCL specifications. In: Proceedings of the Conference on Design, Automation and Test in Europe. DATE '13, EDA Consortium (2013) 1189–1192
[28] Clarisó, R., González, C.A., Cabot, J.: Towards domain refinement for UML/OCL bounded verification. In: Software Engineering and Formal Methods: 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings. Springer International Publishing, Cham (2015) 108–114
[29] Maraee, A., Balaban, M.: Removing redundancies and deducing equivalences in UML class diagrams. In: Model-Driven Engineering Languages and Systems. Volume 8767 of Lecture Notes in Computer Science. Springer (2014) 235–251
[30] Wahler, M., Basin, D., Brucker, A.D., Koehler, J.: Efficient analysis of pattern-based constraint specifications. Software & Systems Modeling **9**(2) (2010) 225–255

Appendix A. Theorem Proofs

In the following we formally prove that the theorems for deleting vertices we have introduced in section 5 preserve the satisfiability of the goal. The proof consists of three steps:

1. We formally characterize the E_{RC} edges. That is, we formalize the idea that there is an edge from a repair vertex to a constraint vertex if we might violate the latter when instantiating the former.
2. We prove that the previous characterization is satisfied in the graph built by Algorithm 1, and that it remains satisfied after the deletion of *irrelevant edges* according to Conditions 1, 2, 3, and 4.
3. We prove that Theorems 2, 3, and 4 for deleting vertices are valid in any dependency graph satisfying the previous characterization.

Now, we review all the steps separately.

Appendix A.1. Defining the Graph Characterization

Consider two *deds* d_1 and d_2 represented by the constraint vertices c_1 and c_2 in a CR-graph. Assume additionally that c_1 has a repair vertex r_1. In the previous sections, we have stated that there is an edge from r_1 to c_2 if repairing the *ded* d_1 by means of r_1 might end violating the *ded* d_2.

In the following, we formalize this idea. To do so, we consider two atoms $a_{r_1} \in f(r_1)$ and $a_{c_2} \in f(c_2)$ with the same predicate and the substitution θ s.t. $a_{r_1} = a_{c_2}\theta$.

Characterization. *There is an edge* $<a_{r_1},a_{c_2}>$ *from* r_1 *to* c_2 *if there is an instantiation* I*, together a ground substitution* σ_1 *for the variables in* $f(c_1)$ *s.t:*

$$I \not\models d_1\sigma_1, \text{ and}$$

$$\exists\sigma_{r_1}.\ I' \models d_1\sigma_1 \wedge I' \not\models d_2\theta\sigma_1\sigma_{r_1}, \text{ and}$$

$$\forall\sigma_{r_x}\exists\sigma_{r_1}.\ I'_x \models d_1\sigma_1 \implies \exists d_i, \theta_i(I'_x \not\models d_i\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1})$$

where I' *stands for* $I \cup f(r_1)\sigma_1\sigma_{r_1}$*,* I'_x *for* $I \cup f(r_1)\sigma_1\sigma_{r_x}\sigma_{r_1}$

Roughly speaking, the first two lines of the previous characterization says that there is an edge between r_1 and c_2 if there is an instantiation that violates d_1 and one instantiation of $f(r_1)$ that repairs d_1 but violating d_2. Then, the last line tightens the characterization to ensure that there is no special value to instantiate an existential variable in $f(r_1)$ s.t. avoids violating any dependency affected by that choice.

Sometimes, it might be useful to consider the contraposition of this characterization, i.e.:

Characterization. *(contraposition) If there is no edge* $<a_{r_1},a_{c_2}>$ *from* r_1 *to* c_2*, then, for any instantiation* I*, and ground substitution* σ_1 *for the variables in* $f(c_1)$*, the following occurs:*

$$I \models d_1\sigma_1, \text{ or}$$

$$\forall\sigma_{r_1}.\ I' \models d_1\sigma_1 \implies I' \models d_2\theta\sigma_1\sigma_{r_1}, \text{ or}$$

$$\exists\sigma_{r_x}\forall\sigma_{r_1}.\ I'_x \models d_1\sigma_1 \wedge \forall d_i, \theta_i(I'_x \models d_i\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1})$$

where I' *stands for* $I \cup f(r_1)\sigma_1\sigma_{r_1}$*, and* I'_x *for* $I \cup f(r_1)\sigma_1\sigma_{r_x}\sigma_{r_1}$*.*

Intuitively, the contraposition says that if there is no edge between r_1 and c_2, then, either it is impossible to violate c_2 by means of instantiating $f(r_1)$, or there is a substitution σ_{r_x} for some concrete existential variable x appearing in a_{r_1} s.t. avoids the violation of any dependency d_i (including d_2) affected by the value given to x.

Appendix A.2. Proving that the Characterization is Satisfied

Since Algorithm 1 adds all the possible E_{RC} edges, the initial graph built by this algorithm trivially satisfies the characterization.

Now, we show that the previous characterization is still satisfied after the elimination of *irrelevant edges*. For doing so, we consider the characterization contraposition.

We review each condition for detecting *irrelevant edges* separately. For doing so, we assume the same structure we have shown in Figure 7.

Appendix A.2.1. Condition 1

Any irrelevant edge detected by Condition 1 satisfies that, for any instantiation I and ground substitution σ_1, we have $I \models d_1\sigma_1 \vee \forall\sigma_{r_1}.\ I' \models d_1\sigma_1 \implies I' \models d_2\theta\sigma_1\sigma_{r_1}$. Thus, removing this edge still satisfies the characterization. We prove so in the following lines:

Proof. Consider an arbitrary instantiation I, a ground substitution σ_1 s.t. $I \not\models d_1\sigma_1$, and any instantiation I' resulting from instantiating $f(r_1)\sigma_1$ in I s.t. $I' \models d_1\sigma_1$. Since $I \not\models d_1\sigma_1$, by definition we have that $I \models f(c_1)\sigma_1$. Moreover, by construction of I' we have that $I' \models (f(c_1) \cup f(r_1))\sigma_1\sigma_{r_1}$, for some σ_{r_1}. Since the condition that permits identifying the edge as irrelevant ensures that there exists a repair r_2 for c_2 together a substitution θ_2 s.t. $f(r_2)\theta\theta_2 \subseteq f(c_1) \cup f(r_1)$, we have that $I' \models f(r_2)\theta\theta_2\sigma_1\sigma_{r_1}$. Since the condition also ensures that $(dom(\sigma_1) \cup dom(\sigma_{r_1})) \cap dom(\theta_2) = \emptyset$, $I' \models f(r_2)\theta\sigma_1\sigma_{r_1}\theta_2\sigma_1\sigma_{r_1}$. Lets now define $\sigma_{r_2} = \theta_2\sigma_1\sigma_{r_1}$. In this manner, $I' \models f(r_2)\theta\sigma_1\sigma_{r_1}\sigma_{r_2}$. Thus, $I' \models d_2\theta\sigma_1\sigma_{r_1}$ because one of its repairs ($f(r_2)$) is already instantiated. □

Appendix A.2.2. Condition 2

Similarly as before, any irrelevant edge detected by Condition 2 satisfies that, for any instantiation I and ground substitution σ_1, we have $I \models d_1\sigma_1$ or $\forall\sigma_{r_1}.\ I' \models d_1\sigma_1 \implies I' \models d_2\theta\sigma_1\sigma_{r_1}$. Thus, removing this edge still satisfies the characterization. We prove so in the following lines:

Proof. Consider an arbitrary instantiation I, a ground substitution σ_1 s.t. $I \not\models d_1\sigma_1$, and any instantiation I' resulting from instantiating $f(r_1)\sigma_1$ in I with the substitution σ_{r_1} s.t. $I' \models d_1\sigma_1$. We will show that $I' \models d_2\theta\sigma_1\sigma_{r_1}$ by contradiction.

Suppose $I' \not\models d_2\theta\sigma_1\sigma_{r_1}$. Thus, by definition, $I' \models f(c_2)\theta\sigma_1\sigma_{r_1}$. Since the condition that permits identifying the edge as irrelevant ensures that there exists a substitution θ_2 s.t. $f(r_1)\theta_2 \subseteq f(c_2)\theta \setminus f(r_1)\theta_2'$, for any substitution θ_2' from terms of $f(r_1)$ to terms of $f(c_2)$, we see that even taking out $f(r_1)\sigma_1\sigma_{r_1}$ from I', I' would still satisfy $f(r_1)\theta_2\sigma_1\sigma_{r_1}$. Recall now that I' without $f(r_1)\sigma_1\sigma_{r_1}$ is precisely the original instantiation I. Thus, we reach the conclusion that $I \models f(r_1)\theta_2\sigma_1\sigma_{r_1}$. Since the condition ensures that $dom(\theta_2) \cap dom(\sigma_1) = \emptyset$, $I \models f(r_1)\sigma_1\theta_2\sigma_1\sigma_{r_1}$. Lets now define $\sigma'_{r_1} = \theta_2\sigma_1\sigma_{r_1}$. In this manner, $I \models f(r_1)\sigma_1\sigma'_{r_1}$. Thus, $I \models d_1\sigma_1$ because its repair r_1 is already instantiated, which contradicts our initial supposition that $I \not\models d_1\sigma_1$. □

Appendix A.2.3. Condition 3

Any irrelevant edge detected by Condition 3 satisfies that, for any instantiation I and ground substitution σ_1, we have $I \models d_1\sigma_1$ or $\exists\sigma_{r_x}\forall\sigma_{r_1}.\ I'_x \models d_1\sigma_1 \wedge \forall d_i, \theta_i(I'_x \models d_i\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1})$. Thus, removing this edge still satisfies the characterization. We prove so in the following lines:

Proof. Consider an arbitrary instantiation I, a ground substitution σ_1 s.t. $I \not\models d_1\sigma_1$, and an instantiation I'_x resulting from instantiating $f(r_1)\sigma_1$ in I with a substitution σ_{r_x} for the variable x identified in the condition, bringing a new fresh value to it. Note that this new fresh value exists because the domain of x is infinite. Clearly, $I'_x \models d_1\sigma_1$. We now show $\forall d_i, \theta_i(I'_x \models d_i\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1})$ by contradiction.

Suppose some d_i and θ_i such that $I'_x \not\models d_i\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1}$. By definition, we have $I'_x \models f(c_i)\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1}$. This means that, in this case, I'_x contains, at least, $k_{min} = instancesLowerBound(f(c_i)\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1}, p, x\sigma_{r_x})$ instances of predicate p with the value $x\sigma_{r_x}$.

Now consider the original instantiation I. Since $x\sigma_{r_x}$ is a new fresh value not present in I, I contains no instances of predicate p with the value $x\sigma_{r_x}$. Thus, the number of instances of predicate p containing the value $x\sigma_{r_x}$ in I'_x is determined by the new instances created when instantiating $f(r_1)$, thus, it can be upper bound by $k_{max} = instancesUpperBound(f(r_1)\sigma_1\sigma_{r_x}\sigma_{r_1}, p, x\sigma_{r_x})$. Thus, $k_{min} <= k_{max}$, in order to get the violation of d_2.

However, the condition ensures $instancesUpperBound(f(r_1), p, x) < instancesLowerBound(f(c_i)\theta_i, p, x)$. Thus, the condition also implies that the number $instancesUpperBound(f(r_1)\sigma_1\sigma_{r_x}\sigma_{r_1}, p, x\sigma_{r_x})$ should be lower than the number $instancesLowerBound(f(c_i)\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1}, p, x\sigma_{r_x})$. That is, $k_{min} > k_{max}$, which is a contradiction. □

Appendix A.2.4. Condition 4

Similarly as before, any irrelevant edge detected by Condition 4 satisfies that, for any instantiation I and ground substitution σ_1, we have $I \models d_1\sigma_1 \vee \exists\sigma_{r_x}\forall\sigma_{r_1}.\ I'_x \models d_1\sigma_1 \wedge \forall d_i, \theta_i(I'_x \models d_i\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1})$. Thus, removing this edge still satisfies the characterization. We prove so in the following lines:

Proof. Consider an arbitrary instantiation I, a ground substitution σ_1 s.t. $I \not\models d_1\sigma_1$, and an instantiation I' resulting from instantiating $f(r_1)\sigma_1$ in I considering a substitution σ_{r_x} for x (the variable identified by the condition), and an arbitrary substitution σ_{r_1} for the rest of variables, s.t. σ_{r_x} brings a value to x lower (or greater, respectively) than any other value in I and any other different value brough by σ_{r_1}. Additionally, consider that we build σ_{r_1} in such a way that only those variables for which we apply this condition are mapped to this lowest (greatest) value. Clearly, $I'_x \models d_1\sigma_1$. We now show $\forall d_i, \theta_i(I' \models d_i\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1})$ by contradiction. For the seek of simplicity, we show the proof for the case in which the built-in literals restricts the minimum value. The case restricting the maximum value is almost identical changing the arithmetic comparisons accordingly.

Suppose some d_i and θ_i s.t. $I'_x \not\models d_i\theta_i\sigma_1\sigma_x\sigma_{r_1}$. By definition, we have that $I'_x \models f(c_i)\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1}\sigma_2$ for some σ_2. Because of the condition, we know for certain that a built-in literal with the form $x\sigma_{r_x} > t\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1}\sigma_2$ (or $\geq$) belongs to $f(c_i)\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1}\sigma_2$, thus, $I'_x \models x\sigma_{r_x} \geq t\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1}\sigma_2$. The value $t\theta_i\sigma_1\sigma_{r_x}\sigma_{r_1}\sigma_2$ present in I'_x comes from either I or from the substitution σ_{r_1}, or from the substitution σ_x in case $t\theta_i$ is x. In the first two cases, the value $x\sigma_{r_x}$ is lower than the other by construction, which is a contradiction. The last case cannot occur because then, the variable x would appear in a built-in literal restricting both: its minimum and and its maximum value; fact that contradicts the condition statement. □

Appendix A.3. Proving the Theorems for Removing Constraint Vertices

Clearly, given a set of *deds* D and some proper subset of it D', if D is satisfiable, then D' is also satisfiable. This is because for any instantiation I, if $I \models D$, then $I \models D'$. Therefor, it is clear that if the original set of *deds* is satisfiable, then, its simplified set is also satisfiable.

Thus, in the following we prove the converse, i.e., when removing some constraint because of theorems 2, 3, and 4, if the simplified set of *deds* is satisfiable, then, the original set of *deds* is satisfiable too.

Appendix A.3.1. Theorem 2

Proof. Given a set of *deds* D, and the set of *deds* D' corresponding to delete from D all those *deds* according to Theorem 2, if D' is satisfiable, then, D is satisfiable.

Suppose that D' is satisfiable. Then, suppose I' to be a minimal instantiation that satisfies D' (i.e., assume that no proper subset of I' satisfies D').

If I' satisfies all *ded* $d \in D \setminus D'$, then, I' witnesses that D is also satisfiable.

Otherwise, consider some *ded* $d \in D \setminus D'$ s.t. $I' \not\models d$ and let c be its constraint vertex. Then, for some ground substitution σ, $I' \models f(c)\sigma$. Now, pick some atom a from $f(c)$. We know that $a\sigma \in I'$. Consider now $I'' = I' \setminus \{a\sigma\}$.

If $I'' \models D'$, then, we rich a contradiction since I' would not be minimal.

Otherwise, consider some $d' \in D'$ s.t. $I'' \not\models d'$. Clearly, for any d', $I' \models d'$ since $I' \models D'$. Thus, adding $a\sigma$ in I' to obtain I'' repairs this d'.

Then, two possibilities arise. One possibility is that there is some edge from some repair vertex r to c. However, since r is a repair vertex from D', and every vertex in D' has a path from some constraint vertex c_0 where $f(c_0) = \top$, there would be a a path from c_0 to c, which contradicts the characterization for removing c.

The other possibility is that there is no edge from r to c. In this case, according to the characterization of the edges, there should exist a substitution σ_{r_x} for some variable x such that repairs d' without violating d. Using this σ_{r_x} we can build a new I_i that avoids the violation of d. Thus, D is satisfiable too. □

Appendix A.3.2. Theorem 3

Proof. Given a set of *deds* D, and the set of *deds* D' corresponding to delete some *ded* d from D according to Theorem 3, if D' is satisfiable, then, D is satisfiable.

Suppose that D' is satisfiable. Then, suppose I' to be a minimal instantiation that satisfies D' (i.e., assume that no subset of I' satisfies D'). We are now going to show that $I' \models D$.

Since I' satisfies any *ded* in D', we only lack to prove that $I' \models d$. We will do it by contradiction.

Suppose $I' \not\models d$. Then, for some ground substitution σ, $I' \models f(c)\sigma$. Because of the characterization of c according to the theorem, $f(c)$ contains some atom a with predicate p s.t. no atom of any repair vertex r contains an atom of p. Thus, consider now I'' as I' after removing any instance of p.

If $I'' \models D'$, then, we rich a contradiction since I' would not be minimal.

If $I'' \not\models D'$, then, there exists $d' \in D'$ s.t. $I'' \not\models d'$. Clearly, $I' \models d'$ because $I' \models D'$. Thus, adding instances of p in I'' to obtain I' repairs d'. Hence, there should be a repair vertex with some atom of predicate p. However, this contradicts the characterization for removing c. □

Appendix A.3.3. Theorem 4

Proof. Given a set of *deds* D, and the set of *deds* D' corresponding to delete some *ded* d from D according to Theorem 4, if D' is satisfiable, then, D is satisfiable.

Suppose D' is satisfiable. Then, there exists some instantiation I' s.t. $I' \models D'$.

If $I' \models d$, then, $I' \models D$ and the claim is proven. Otherwise, if $I' \not\models d$, then, for some ground substitution σ, $I' \models f(c)\sigma$. Note that there might be more than one σ. Then, for each σ, we can define I_i as I' after adding $f(r)\sigma\sigma_r$ for some ground substitution σ_r. Thus, we repair any violation of d in I'.

Since r does not point to any other constraint vertex, this means that by such insertions, either we do not violate any other *ded* d', or that there is a special substitution σ_{r_x} for some variable x in $f(r)$ s.t. repairs the constraint without violating d' neither any other dependency affected by the value given to x. Thus, we can build some I_i s.t. $I_i \models D$. □