Graph Rewriting System for Consistent Evolution of RDF/S databases

Jacques Chabin, Cédric Eichler, Mirian Halfeld-Ferrari, Nicolas Hiot

Graph Rewriting System for Consistent Evolution of RDF/S databases *

Jacques Chabin, Cédric Eichler, Mirian Halfeld-Ferrari, and Nicolas Hiot

Université d'Orléans, INSA CVL, LIFO EA, Orléans, France
{jacques.chabin, mirian, nicolas.hiot}@univ-orleans.fr
cedric.eichler@insa-cvl.fr

Abstract. This report investigates the use of graph rewriting rules to model updates -instance or schema changes- on RDF/S databases which are expected to satisfy RDF intrinsic semantic constraints. Such databases being modeled as knowledge graphs, we propose graph rewriting rules formalizing atomic updates whose application transforms the graph and necessarily preserves its consistency.
If an update has to be applied when the application conditions of the corresponding rule do not hold, side-effects are generated: they engender new updates in order to ensure the rule applicability. Our system, SetUp, implements our updating approach for RDF/S data and offers a theoretical and applied framework for ensuring consistency when a RDF knowledge graph evolves.

Keywords: Graph rewriting · Updates · Constraints · RDF.

1 Introduction

Today RDF (Resource Description Framework) is a standard model for data interchange on the Web and particularly for exporting Linked Open Data. These data, enriched by constraints, stored in a database (especially in a graph database or triple store), and available for querying systems are important sources for analysis and for guiding decisions. But only systems offering the capability of dealing with evolution of data instance and structure without violating the semantics of the RDF model can ensure sustainability. Since RDF/S database can be canonically viewed as knowledge graph, this paper proposes the use of *graph rewriting techniques* in the evolution of RDF data stores, showing the utility of this formal tool into a practical and useful application. RDF/S is the focus of the paper because, currently, users interested in these facilities are mostly those dealing with ontology evolution. However, new graph applications have now increasing demands concerning their validity with respect to a set of constraints. Our proposal naturally adapts to these new contexts.

We are interested in the maintenance of *valid* RDF/S knowledge graph, *i.e.*, data sets respecting constraints. When such a data set evolves (through instance

* Supported by the French National Research Agency, ANR, under grant ANR-18-CE23-0010; work also developed in the context of DOING-DIAMS network group.

or schema changes) we have to guarantee that the new set conforms to the constraints. The following example illustrates the problem.

Example 1 (Motivating Example.). Fig. 1 shows a *complete* RDF/S graph database consistent *w.r.t.* to the RDF/S semantic constraints. We are concerned by the problem of updating this database, keeping it consistent. Firstly, suppose an instance update: the insertion of a *ASA* as a class instance of *Molecule*. How can we guarantee that *ASA* will be also an instance all the super-classes of *Molecule*? Then, consider a schema evolution: the insertion of *provokeReaction* as sub-property of *HasConsequence*. How can we perform this change ensuring that *provokeReaction* will have its domain and range as sub-classes of those of *HasConsequence*? □

This paper proposes SetUp (Schema Evolution Through UPdates), a maintenance tool for RDF knowledge graph.

SetUp summarized in two main steps
(1) Firstly we formalize updates as *graph rewriting rules* encompassing integrity constraints. An *Update* is a general term and can be classified through two different aspects: on one hand, as insertions or deletions and, on the other hand as instance or schema changes. Each update is formalized by a graph rewriting rule whose application *necessarily* preserves the databases validity. To perform an update, the applicability conditions of the corresponding rule are automatically checked. When all conditions of a rule hold, the rule is activated to produce a new graph which takes into account the required update and is necessarily valid if the graph was valid prior to the update. The use of graph rewriting rules ensures consistency preservation *in design time* – no further verification is needed in runtime.
(2) Secondly, if the applicability condition of a rule *does not hold*, the update is rejected. SetUp provides the possibility to force its (valid) application by performing *side-effects*. Indeed, in our method, side-effects are new updates that should be performed to allow the satisfaction of a rule's condition. Side-effects are implemented by procedures associated to an update type, and thus, to some rewriting rules. When an evolution is mandatory, we enforce database evolution by performing *side effects* (*i.e.*, triggering other updates or schema modifications which will render possible rule application).

SetUp's main characteristics.
• SetUp main goal is to ensure validity when dealing with the evolution of an RDF/S knowledge graph. Such a graph represents a set of RDF (the instance) and RDFS (the schema) triples which respect semantic constraints as defined in [5].
• SetUp deals with *complete* instances, *i.e.*, constraint satisfaction is obtained only when the required data is *effectively* stored in the database.
• SetUp implements deterministic rules. Arbitrary choices have been made when non-deterministic options are available.
• SetUp takes into account the user level. Only database administrators may require updates provoking schema changes.

Paper Organization. Sections 2 and 3 offer useful definitions and formal background on graph rewriting systems. Updates are formalized by rewriting rules in Section 4 and side-effects considered in Section 5. Section 6 briefly discusses different implementation options and Sec. 7 presents experimental results obtained with our current prototype. Related work and concluding remarks are given in Sections 8 and 9.

2 RDF databases and updates

The RDF data model describes (web) resources via triples of the format $(a\ P\ b)$, which express the fact that a has b as value for property P. A collection of RDF statements intrinsically represents a typed attributed directed multi-graph, making the RDF model suited to certain kinds of knowledge representation [1]. Constraints on RDF facts can be expressed in RDFS (Resource Description Framework Schema), the schema language of RDF, which allows, for instance, declaring objects and subjects as instances of certain classes or expressing semantic relations between classes and between properties (*i.e.*, subclasses and sub-properties).

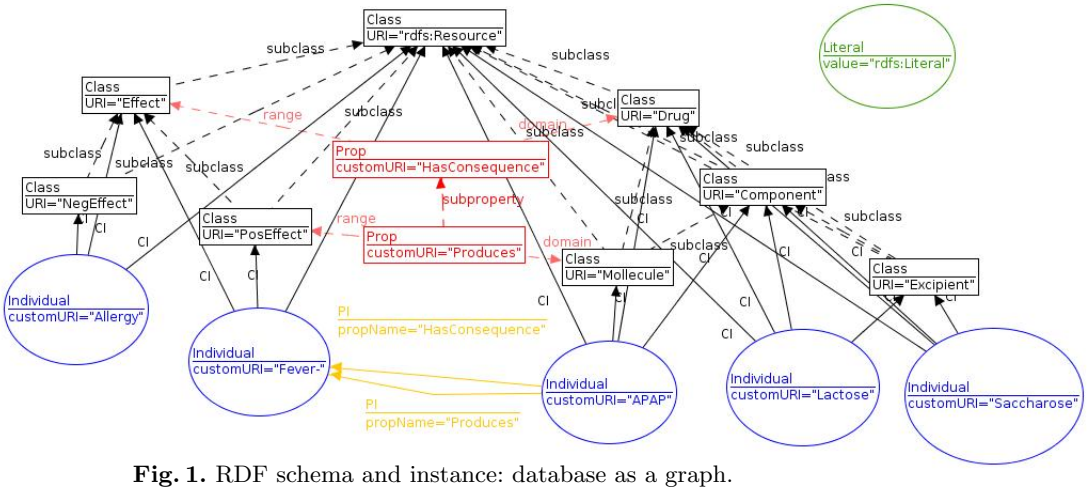


Fig. 1. RDF schema and instance: database as a graph.

In [5] we find a set of logical rules expressing the semantics of RDF/S (rules concerning RDF or RDFS) models. We consider $\mathbf{A}_C = \{a, b, \ldots, a_1, a_2, \ldots\}$, a countably infinite set of constants and $var = \{X_1, X_2, \ldots, Y_1, \ldots\}$ an infinite set of variables ranging over elements in $\mathbf{A}_C$. A *term* is a constant or a variable. We classify predicates into two sets: (i) SchPred $= \{Cl, Pr, CSub, Psub, Dom, Rng\}$, used to define the database schema, standing respectively for classes, properties, sub-classes, sub-properties, property domain and range, and (ii) InstPred $= \{CI, PI, Ind\}$, used to define the database instance, standing respectively for class and property instances and individuals. An *atom* has the form $P(u)$, where

Schema specification:
$Cl(Drug)$; $Cl(Component)$; $Cl(Excipient)$; $Cl(Mollecule)$; $Cl(Effect)$; $Cl(PosEffect)$;
$Cl(NegEffect)$; $Pr(HasConsequence)$; $Pr(Produces)$; $CSub(Drug, rdfs:Resource)$;
$CSub(Component, rdfs:Resource)$; $CSub(Excipient, rdfs:Resource)$;
$CSub(Mollecule, rdfs:Resource)$; $CSub(Effect, rdfs:Resource)$; $CSub(PosEffect, rdfs:Resource)$;
$CSub(NegEffect, rdfs:Resource)$; $CSub(Component, Drug)$; $CSub(Excipient, Drug)$;
$CSub(Mollecule, Drug)$; $CSub(Excipient, Component)$; $CSub(Mollecule, Component)$;
$CSub(PosEffect, Effect)$; $CSub(NegEffect, Effect)$; $Dom(HasConsequence, Drug)$;
$Rng(HasConsequence, Effect)$; $Dom(Produces, Mollecule)$; $Rng(Produces, PosEffect)$;
$PSub(Produces, HasConsequence)$;
Database instance:
$Ind(Saccharose)$; $Ind(Lactose)$; $Ind(APAP)$; $Ind(Fever^-)$; $Ind(Allergy)$;
$CI(Saccharose, rdfs:Resource)$; $CI(Saccharose, Drug)$; $CI(Saccharose, Component)$;
$CI(Saccharose, Excipient)$; $CI(Lactose, rdfs:Resource)$; $CI(Lactose, Drug)$;
$CI(Lactose, Component)$; $CI(Lactose, Excipient)$; $CI(APAP, rdfs:Resource)$; $CI(APAP, Drug)$;
$CI(APAP, Component)$; $CI(APAP, Mollecule)$; $CI(Fever^-, rdfs:Resource)$; $CI(Fever^-, Effect)$;
$CI(Fever^-, PosEffect)$; $CI(Allergy, rdfs:Resource)$; $CI(Allergy, Effect)$; $CI(Allergy, NegEffect)$;
$PI(APAP, Fever^-, HasConsequence)$; $PI(APAP, Fever^-, Produces)$;

Fig. 2. RDF schema and instance (logical point of view): database as a set of facts.

P is a predicate, and u is a list of terms. When all the terms of an atom are in $\mathbf{A}_C$, we have a fact. Figure 1 shows an RDF instance and schema as a typed graph whose specification are given in Appendix A. Figure 2 shows the same database using its logical representation as positive atoms. Class *"rdfs:Resource"* symbolizes the root of an RDF class hierarchy.

Node and edge are typed (represented with a unique combination of shape and color, *e.g.* , an individual node is a blue oval) and attributed to indicate a name or value when relevant. For instance, in Fig. 1 the schema specifies that *Has Consequence* is a property having class *Drug* as its domain and the class *Effect* as its range. The property *Produces* is a sub-property of *Has Consequence* while *PosEffect* is a sub-class of *Effect*. Database instance is represented by individuals which are elements of a class (*e.g.* $APAP$ is an instance of class *Mollecule*) and their relationships (*e.g.* the property instance *Produces*, between $APAP$ and $Fever^-$).

Definition 1 (Database). *An RDF database $\mathscr{D}$ is a set of facts composed by two subsets: the database instance $\mathscr{D}_I$ (facts with predicates in* InstPred*) and the database schema $\mathscr{D}_S$ (facts with predicates in* SchPred*). Denote by $\mathbb{G}$ the graph that represents the same database $\mathscr{D}$. Let $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ where $\mathbb{V}$ are nodes with type in $\{Cl, Pr, Ind, Lit\}$ and $\mathbb{E}$ are edges having type in $\{Dom, Rng, PSub, CSub, CI, PI\}$. The notation $\mathscr{D}/\mathbb{G}$ designates these two formats of a database.* □

Constraints presented in [5] are those in Fig. 3 which is borrowed from [11]. These constraints (that we denote by $\mathscr{C}$) are the basis of RDF semantics. Given $c \in \mathscr{C}$ we note $body(c)$ its left-hand side and $head(c)$ its right-hand side. For instance, the schema constraint (20) establishes transitivity between sub-properties and the instance constraint (27) ensures this transitivity on instances of a property (if z is a sub-property of w, all z's instances are property instances of w). We are interested in database that satisfy all constraints in $\mathscr{C}$.

Definition 2 (Consistent database $(\mathscr{D}, \mathscr{C})$). *A database $\mathscr{D}$ is consistent if it satisfies all constraints in $\mathscr{C}$ (i.e., in this paper, those in Fig. 3).* □

• **Typing Constraints:**			
$Cl(x) \Rightarrow URI(x)$	(1)	$Pr(x) \Rightarrow URI(x)$	(2)
$Ind(x) \Rightarrow URI(x)$	(3)	$(Cl(x) \wedge Pr(x)) \Rightarrow \bot$	(4)
$(Cl(x) \wedge Ind(x)) \Rightarrow \bot$	(5)	$(Pr(x) \wedge Ind(x)) \Rightarrow \bot$	(6)
$CSub(x,y) \Rightarrow Cl(x) \wedge Cl(y)$	(7)	$PSub(x,y) \Rightarrow Pr(x) \wedge Pr(y)$	(8)
$Dom(x,y) \Rightarrow Pr(x) \wedge Cl(y)$	(9)	$Rng(x,y) \Rightarrow Pr(x) \wedge Cl(y)$	(10)
$CI(x,y) \Rightarrow Ind(x) \wedge Cl(y)$	(11)	$PI(x,y,z) \Rightarrow Ind(x) \wedge (Ind(y) \vee Lit(y)) \wedge Pr(z)$	(12)
$Cl(x) \Rightarrow CSub(x, rdfs{:}Resource)$	(13)	$Ind(x) \Rightarrow CI(x, rdfs{:}Resource)$	(14)
• **Schema Constraints:**			
$Pr(x) \Rightarrow (\exists y,z)(Dom(x,y) \wedge Rng(x,y))$	(15)	$((y \neq z) \wedge Dom(x,y) \wedge Dom(x,z)) \Rightarrow \bot$	(16)
$((y \neq z) \wedge Rng(x,y) \wedge Rng(x,z)) \Rightarrow \bot$	(17)		
$CSub(x,y) \wedge CSub(y,z) \Rightarrow CSub(x,z)$	(18)	$CSub(x,y) \wedge CSub(y,x) \Rightarrow \bot$	(19)
$PSub(x,y) \wedge PSub(y,z) \Rightarrow PSub(x,z)$	(20)	$PSub(x,y) \wedge PSub(y,x) \Rightarrow \bot$	(21)
$Psub(x,y) \wedge Dom(x,z) \wedge Dom(y,w) \wedge (z \neq w) \Rightarrow CSub(z,w)$			(22)
$Psub(x,y) \wedge Rng(x,z) \wedge Rng(y,w) \wedge (z \neq w) \Rightarrow CSub(z,w)$			(23)
• **Instance Constraints:**			
$Dom(z,w) \Rightarrow (PI(x,y,z) \Rightarrow CI(x,w))$	(24)	$Rng(z,w) \Rightarrow (PI(x,y,z) \Rightarrow CI(x,w))$	(25)
$CSub(y,z) \Rightarrow (CI(x,y) \Rightarrow CI(x,z))$	(26)	$PSub(z,w) \Rightarrow (PI(x,y,z) \Rightarrow PI(x,y,w))$	(27)

Fig. 3. Simplified and compacted form of RDF/S constraints

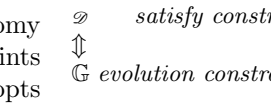


Fig. 4. Rewriting rules $\mathbb{R}$ and constraints $\mathscr{C}$.

At this point, it is worth noting the dichotomy which usually exists when dealing with constraints on RDF. The web semantics world mostly adopts the open world assumption (OWA) and ontological constraints are, in fact, just inference rules. The database world usually adopts the closed world assumption (CWA) and constraints impose data restrictions. Rules are not supposed to infer a non-explicit knowledge. This paper adopts the database point of view and addresses the problem of updating an RDF database.

Definition 3 (Update). *Let $\mathscr{D}/\mathbb{G}$ be a database. An update U on $\mathscr{D}$ (for $U = F$) is either (i) the insertion of F in $\mathscr{D}$ (an insertion is denoted by F) or (ii) the removal of F from $\mathscr{D}$ (a deletion is denoted by $\neg F$). To each update U corresponds a graph rewriting rule r.* □

Updates can be classified according to the predicate of F, *i.e.*, the insertion (or the deletion) of a class, a class instance, a property, etc. For each update type, a rewriting rule r describes when and how to transform a graph database. This paper aims at proposing a set of graph rewriting rules, denoted by $\mathbb{R}$, which ensures consistent transformations on $\mathbb{G}$ due to any atomic update U. The set $\mathbb{R}$ is defined on the basis of $\mathscr{C}$ as illustrated in Fig. 4: on the logical level, $(\mathscr{D}, \mathscr{C})$ expresses consistent databases; on the knowledge graph level, $(\mathbb{G}, \mathbb{R})$ expresses graph evolution with rules in $\mathbb{R}$ encompassing constraints from $\mathscr{C}$. The idea is: given $\mathscr{D}/\mathbb{G}$ for $(\mathscr{D}, \mathscr{C})$ and update U corresponding to rule $r \in \mathbb{R}$; if $\mathbb{G}$' is the result of applying r on $\mathbb{G}$ then our goal is to have $(\mathscr{D}', \mathscr{C})$ for $\mathscr{D}'/\mathbb{G}'$.

3 Preliminaries: graph rewriting

We briefly introduce the theoretical background on the formal graph rewriting approach adopted in this paper: the Single Push Out (SPO) [12] approach.

Graph rewriting is a well-studied field for the formal specification of graph transformations [16]. It relies on the definition of graph rewriting rules which specify both the effect of a graph transformation and the context in which it may be applied. In this paper, we adopt the SPO formalism [12] to specify rewriting rule as well as several extensions of its extension to specify additional application conditions and restrict rule applicability: *Negative Application Conditions* (NACs) [8], *Positive Application Conditions* (PACs), and *General Application Conditions* (GACs) [17].

Specifying rewriting rules using the SPO approach. The SPO approach is an algebraic approach based on category theory. A rule is defined by two graphs – the Left and Right Hand Side of the rule, denoted by L and R or LHS and RHS – and a partial morphism m from L to R. An example of an SPO rule is specified in Fig. 5. The LHS of the rule is composed by a single node of type *Class* whose *Type* attribute is set to *"rdfs:Resource"*. The RHS of the rule is composed by two *Class* nodes with attribute values *"rdfs:Resource"* and A and an edge of type *Subclass* from the latter to the former. By convention, an attribute value within quotation mark (e.g. *"rdfs:Resource"*) is a fixed constant, while a value noted without quotation mark (e.g. A) is a variable whose value may be given as an input or assigned according to the context.

The partial morphism from L to R is specified in the figure by tagging graph elements - nodes or edges - in its domain and range with a numerical value. An element with value i in L is part of the domain of m and its image by m is the graph element in R with the same value i. In the example, the notation *1:* before the node type of the two nodes symbolizing the root of the class hierarchy in L and R indicates that they are mapped through m.

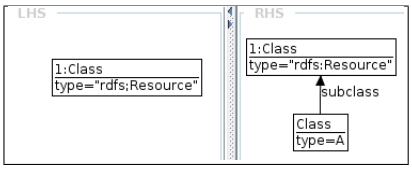


Fig. 5. An SPO rewriting rule

Application of SPO rewriting rules. A graph rewriting rule $r = (L, R, m)$ is applicable to a graph G iff there exists a total morphism $\tilde{m}$ from L to G. The two morphisms $m : L \to R$ and $\tilde{m} : L \to G$ are shown in black in Fig. 6. The object of its push-out, G', depicted in red, is the result of the application of r to G with regard to $\tilde{m}$.

Informally, the application of r to G with regard to $\tilde{m}$ consists in modifying elements of G by (1) removing the image by $\tilde{m}$ of all elements of L that are not in the domain of m (*i.e.*, removing $\tilde{m}(L \backslash Dom(m))$); (2) removing all dangling edges (*i.e.*, deleting all edges that were incident to a node that has been suppressed in step (1)); (3) adding an isomorphic copy of all elements of R that are not in the

domain of m. Going back to the example rule depicted in Fig. 5 this means that the rule is applicable to any graph G containing a class node n with attribute *"rdfs:Resource"*. Its application consists in adding a class node with attribute A and a subclass edge from this node to n. Assuming that A is given as input, this rule is thus a first way of formalizing the addition of a class node to the database. It is however naive since the class node could already be present in the graph, creating a duplicate. To avoid this situation, the applicability of the rule must be further restricted.

Extensions to restrict applicability: NACs and PACs are well-known extensions that forbid or require certain patterns to be present in the graph for a rule to be applicable, respectively. A NAC or a PAC is defined as a constraint graph which is a super-graph of the LHS of the rule they are associated to. An SPO rule $r = (L, R, m)$ with NACs and PACs is applicable to a graph iff: (i) there exists a total morphism $\tilde{m} : L \to G$ (this is the classical SPO application condition); (ii) for all PACs P (resp. NACs N) associated with r, there exists a total morphism (resp. there exists *no* total morphism) $\bar{m} : P \to G$ whose restriction to L is $\tilde{m}$. By convention and to avoid redundancy, since NACs and PACs are super-graphs of L, when illustrating a NAC or a PAC, L will not be depicted. This convention has two major implications. Firstly, it is necessary to explicitly identify graph elements that are common to L and the depicted part of the NAC. This is done similarly to the identification of graph elements matched by the morphism from L to R, by adding numerical value to relevant graph elements in L and NAC/PAC. Secondly, it is important to note that $\tilde{m}$ and $\bar{m}$ are not necessarily injective. However, it is forbidden for an element of the depicted part of the NAC and an element of L to have the same image by $\bar{m}$ in G if they are not explicitly identified as common.

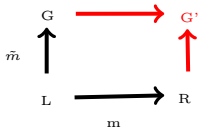


Fig. 6. Push-Out, application of SPO rules

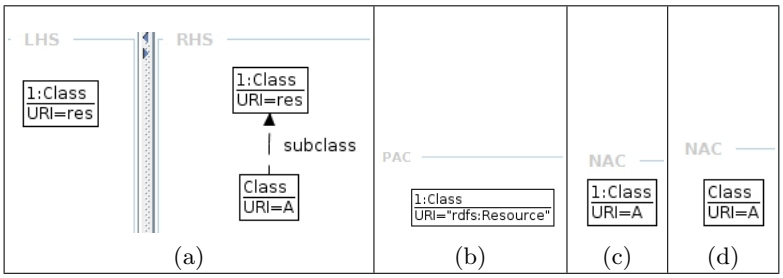


Fig. 7. Less naive rewriting rule for the insertion of a class

Fig. 7 shows an enrichment of the rule depicted in Fig. 5. In the SPO core of the rule, the attribute "$rdfs : Resource$" is simply replaced by a variable res. The PAC specified in Fig. 7b imposes that $res =$ *"rdfs:Resource"*, *i.e.*, the node in L should be the root of the class hierarchy. So far, the rule has the same

behaviour as the one in Fig. 5. In addition, it (i) avoids the addition of duplicate class node, thanks to the NAC of Fig. 7d defined as the juxtaposition of L and a Class attributed $URI{=}A$; (ii) forbids the addition of a second *rdfs:Resource* class node thanks to the NAC presented in Fig. 7c, stating that the input A may not be equal to the res.

The more classical application conditions, be it NACs or PACs, are defined as a constraint graph C and an injective partial morphism (in that case, the identity function) from C to L. That observation lead to the introduction of *nested application conditions* [7,9] that allows to define conditions on the constraint graphs. A condition over a graph G is of the form $true$ or $\exists(a, c)$ where $a : G \to C$ is a graph morphism from G to a condition graph C, and c is a condition over C. With this definition, a PAC P over a rule (L, R, m) can be seen as a condition $(a, true)$, with a being the identity morphism from L to P. Application conditions can be negated, so that a NAC N can be defined as a condition $\neg(a, true)$, with a similar definition of a. GACs [17] are a combination of nested application condition allowing the definition of complex applications conditions for SPO rules. A GAC of a rule (L, R, m) is a condition over L that may be quantified by $\forall$ and combined using $\wedge$ and $\vee$. The rule (L, R, m) with GAC $\exists(a, c)$ is applicable to a graph G with regard to a morphism $\tilde{m}$ if there is an injective graph morphism $\bar{m} : G \to C$ such that $\bar{m} \circ a = \tilde{m}$ and $\bar{m}$ satisfies c.

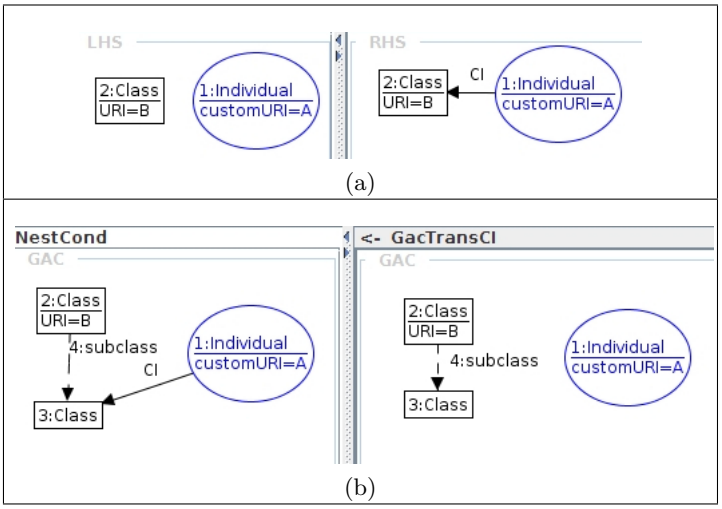


Fig. 8. SPO rule for the insertion of a class instance with a GAC

Fig. 8 shows an example of a rule with a nested GAC of the form $\forall(a, c)$. The morphism a from L to *GacTransCI* is depicted in the right part of Fig. 8b. *GacTransCI* contains L plus a *subclass* edge from the class node of L to a new class node n. The condition c is $\exists(b, true)$, with b the morphism from *GacTransCI* to

NestCond (left part of Fig. 8b): *NestCond* is itself a super-graph of *GacTransCI* and comports one more *CI* edge from the individual node to n. Due to this GAC, the rule is applicable to a graph G with regard to a morphism $\tilde{m}$ only if for all morphism $\bar{m}$ from $GacTransCI$ to G whose restriction to L is $\tilde{m}$, there also exists at least a morphism from $NestCond$ to G which restriction to $GacTransCI$ is $\bar{m}$. In other word, this GAC ensures that if the rule is applicable, then $\forall C, Cl(C) \wedge CSub(B, C) \Rightarrow CI(A, C)$. Indeed, if there is a mapping from L to the database graph, the rule is applicable only if, for each matching of $GacTransCI$ (*i.e.*, for all class C that is a super-class of B) there is a matching of $NestCond$ (*i.e.*, there must be an edge of type CI from $Ind(A)$ to $Cl(C)$).

4 Graph rewriting for consistency maintenance

In our proposal, rewriting rules formalize both graph transformations and the context in which they may be applied. These rules may be *fully specified graphically*, enabling an easy-to-understand graphical view of the graph transformation that remains formal. To prevent the introduction of inconsistencies during updates, we 1) formally specify rules of $\mathbb{R}$ formalizing $\mathbb{G}$ evolution and 2) prove that every rule in $\mathbb{R}$ ensures the preservation of every constraints in $\mathscr{C}$.

Recall from Section 2 the relationships between $\mathscr{D}$ and $\mathbb{G}$ and between $\mathscr{C}$ and $\mathbb{R}$. In this context, we have designed the set $\mathbb{R}$: eighteen graph rewriting rules which formalize atomic updates on $\mathbb{G}$ ensuring database consistent evolution w.r.t. $\mathscr{C}$. The kernel of $\mathbb{R}$'s construction lies on the detection of constraints in $\mathscr{C}$ impacted by an update: an insertion F (respectively, a deletion $\neg F$) impacts constraints having the predicate of F in their body (respectively, in their head). Consider for instance constraint (11): if $CI(A, B)$ is in $\mathscr{D}$ then it should also contain a class B and an individual A. Hence, the graph rewriting rule concerned by the insertion of $CI(A, B)$ is activated only in a database respecting these conditions.

In our approach each update type corresponds to a rule in $\mathbb{R}$. Notice however that two different rules describe the insertion (or the deletion) of a property, depending whether its range is a class or a literal. The 18 rules of $\mathbb{R}$ are available online [2], this section offers the presentation of three of them while the others, together with the proof of their consistency, are presented in Appendix B. The rules considered now are: (a) the insertion of class; (b) the deletion of a class and (c) the insertion of a class instance. Their presentation follows a standard basic form filled by the main explanations of the rule. Recall that the LHS of a SPO specification is related to its applicability while its RHS is related to the effect of its application. The proofs are in fact quite immediate. Indeed, rules have been specified to preserve consistency constraints by-design and their graphical specification, even if formal and respecting the definitions presented in Section 3, ease comprehension.

• Insertion of a Class (Fig.s 7 and 9)
Update category: Schema evolution
User level: Only authorized users such as database administrators

Rule semantics: This rule has been partially presented in Section 3 (Fig. 7).
1) SPO specification: (Fig. 7a)
LHS: there exists a class with URI *res* in the database;
RHS: LHS plus a class with the URI A as a sub-class of *Cl(res)*. The application of the rule will lead to their addition.
2) PAC specification: (Fig. 7b) variable *res* is assigned to "*rdfs:Resource*"; this PAC corresponds to constraint (13).
3)NAC_{res} and NAC_{cl}: (Fig. 7d and 7c respectively) these NACs are non-redundancy guarantee (ie, two classes may not have the same URI). A class *Cl(A)* may be inserted in the graph when: (i) A is not *rdfs:Resource* and (ii) another class with URI A does not already exist.
4) NAC_{ind} and NAC_{pr} (Fig.s 9a and 9b, respectively): guarantee that the sets of classes, properties, and individuals are disjoint (constraints 4 and 5).

Proof of consistency preservation: It is clear from Fig. 3 that the addition of a class may activate constraints 4, 5, and 13 (*i.e.*, those having an atom with predicate Cl in their bodies). Thanks to the specification of NAC_{ind} and NAC_{pr}, constraints 4 and 5 are ensured. The PAC and SPO core of the rule in Fig. 7b and 7a impose the new class to be a subclass of *rdfs:Resource*, as constraint 13.

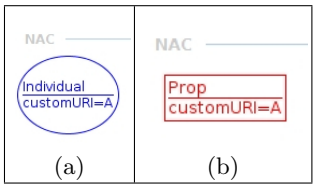


Fig. 9. Additional NACs for addCl(A)

• Deletion of a Class (Fig. 10)
Update category: Schema evolution
User level: Only authorized users such as database administrators
Rule semantics:
1) SPO specification: (Fig. 10a)
LHS: there exist a class with URI A in the database;
RHS: empty, rule's application leads to the deletion of the class with the URI A.
2) NAC_{res} : (Fig. 10b) states that the rule cannot be applied when A is *rdfs:Resource* – indeed the root of RDF class hierarchy cannot be deleted.
3)NAC_{dom} and NAC_{range}: (Fig. 10c and 10d respectively) impose that the class being deleted is neither the domain nor the range of any property.

Proof of consistency preservation: From Fig. 3, the deletion of a class may impact constraints 7, 9, 10, 11 (those having $Cl(A)$ on the RHS) together with constraints 13, 14, 15 (as consequences of possible deletion politics). Constraints 7 and 11 are preserved because $CSub$ and CI relations involving $Cl(A)$ are represented as edge incident to the node modelling $Cl(A)$. As in the SPO approach dangling edges are deleted, all $CSub$ and CI relations involving $Cl(A)$ are suppressed when this rule is applied. Constraint 9 (respect. 10) forbids the deletion of A as the domain (respect. as a range) of an existing property (which would also impact rule 15). Thanks to NAC_{dom} and NAC_{rng}, our graph rewriting rule is applicable only if the class to be deleted is neither the domain nor the range of any property. Finally as NAC_{res} forbids the deletion of class *rdfs:Resource*, constraints 13 and 14 are never violated by the deletion of a class.

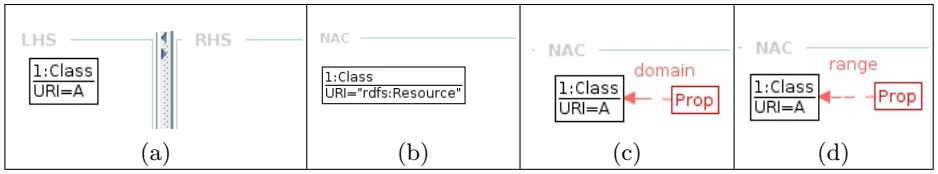


Fig. 10. In (10a) the SPO rule for the deletion of a class, denoted $delCl(A)$. It has 3 NAC, namely (10b) NAC_{res} : $A \neq$ "$rdfs : Resource$"; (10c) NAC_{dom} : $\forall P$ such that $Pr(P)$, $Dom(P, A)$ is false; (10d) NAC_{rng}: $\forall P$ such that $Pr(P)$, $Rng(P, A)$ is false.

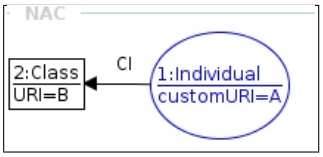


Fig. 11. Additional NAC for addCI(A,B)

• Insertion of a Class Instance (Fig. 8 and 11)
Update category: Instance evolution
User level: Any user
Rule semantics: This rule has been partially presented in Sec. 3 (Fig. 8).
1) SPO specification: (Fig. 8a)
LHS: there exist a class with URI B and an individual A in the database;
RHS: an edge from *Ind(A)* to *Cl(B)* is introduced in the graph.
2) NAC_{red} : (Fig. 11) forbids the application of the rule if $CI(A, B)$ already exists in the database.
3)GAC: (Fig. 8b) ensures that the instance A of class B (being inserted) will also be an instance of all super-classes of B.

Proof of consistency preservation: From Fig. 3, constraints 11 and 26 (having atoms with CI on their body) are impacted. Our graph rewriting rule ensures that the insertion of a class instance is performed only when the individual and its type already exist in the database (constraint 11). According to *GacTrans*, if there exists some super-class C of B and A is not an instance of C, then the class instance relation $CI(A, B)$ cannot be added (ensuring constraint 26).

The following lemma proves that $\mathbb{R}$ ensures the consistency of the database when an update is performed.

Lemma 1 (Correction of rewriting rules). *Let U be an update, F the fact being inserted (resp. deleted) and $r \in \mathbb{R}$ the corresponding rewriting rule. Let $\mathbb{G}/\mathscr{D}$ be a consistent database, $\mathbb{G}$' be the result of the application of r on $\mathbb{G}$ (we write $\mathbb{G}' = r(\mathbb{G})$), and $\mathscr{D}'$ the database defined by $\mathbb{G}'/\mathscr{D}'$. Then (1) $\mathbb{G}'$ is consistent, i.e., ($\mathscr{D}'$,$\mathscr{C}$) and (2) $F \in \mathscr{D}'$ (resp. $F \notin \mathscr{D}'$).* □

Proof. Individual proofs are provided above and in Appendix B for each rule of $\mathbb{R}$. It is shown that each rule preserve the data-base consistency and does indeed add or remove the fact it is related to.

5 Side-effects and Consistent Database Evolution

Traditionally, whenever a database is updated, if constraint violations are detected, either the update is refused or compensation actions, which we call side-effects, must be executed in order to guarantee their satisfaction. In our approach, as explained in Section 4, for each update U there is a rewriting rule

$r_U \in \mathbb{R}$ and the application of r_U relies on whether $\mathbb{G}$ satisfies the premisses of r_U. The graph transformation takes place only when $\mathbb{G}$ respects all the conditions expressed in r_U. If such conditions are not respected, our algorithm generates new updates capable of changing $\mathbb{G}$ into a new graph $\mathbb{G}^n$ on which r_U can be applied to produce $\mathbb{G}'$. These new updates are called side-effects of U.

Example 2. Let $\mathscr{D}/\mathbb{G}$ be the database as the one in Fig. 1, but without the sub-graph concerning *NegEffect*. In this context, consider that U is the insertion $CI(Allergy, NegEffect)$. Let $r_{CI} \in \mathbb{R}$ be the graph rewriting rule concerning the insertion of a class instance (Section 4). Rule r_{CI} cannot be applied on $\mathbb{G}$ since it requires the existence of both the class and the individual which we want to "link together" as class instance. Thus, in this situation, two new updates are generated as *side-effects*:

- U^1 the insertion of an individual *Allergy* and
- U^2 the insertion of class *NegEffect*.

Both updates conditions are checked and, since they are valid, the corresponding rules are triggered, adding the individual and class and connecting them to class *rdfs:Resource*. Once we have the new graph resulting from the application of r_{U^1} and r_{U^2}, rule r_{CI} is applied. The result will be a graph as the one in Fig. 1, except for a missing sub-class edge between *Effect* and *NegEffect* and a missing class instance edge from *Allergy* to *Effect*. □

Roughly, SetUp is an algorithm allowing the interaction between a graph rewriter and a side-effect generator. The latter, producing new updates to be treated by the former, can follow different politics in ordering and in authorizing the treatment of these new updates. Indeed, in our approach, different levels of users are considered: those authorized to trigger side-effects or provoke schema changes and those for whom only instance updates respecting $\mathbb{R}$ are allowed. Algorithm 1 summarizes our approach for *authorized users*.

Update Type	**Conditions**
CI(Xi,XC)	Indiv(Xi); Cl(XC) ∨ (XC = Ressource); ∀ YC CSub(XC, YC) then CI(Xi, YC)
Cl(A)	¬Pr(A); ¬Indiv(A); CSub(A, Resource); Uri(A)
¬ Pr(P)	∀ Xsp PSub(Xsp, P) then ¬ PSub(Xsp, P) ∀ XP PSub(P, XP) then ¬ PSub(P, XP) ∀ XD Dom (P, XD) then ¬ Dom (P, XD) ∀ XR Rng (P, XR) then ¬ Rng (P, XR) ∀ Xi, Yi PI (Xi, Yi, P) then ¬ PI (Xi, Yi, P)

Fig. 12. Extract of UpdCond table.

Given a database $\mathscr{D}/\mathbb{G}$ and an update U, Algorithm 1 transforms $\mathbb{G}$, following rules in $\mathbb{R}$. Denote by $r_U \in \mathbb{R}$ the rewriting rule associated to U. When r_U

cannot be applied on $\mathbb{G}$, SetUp computes, recursively, all updates necessary to change $\mathbb{G}$ into a new graph where r_U applies. Here, it is worth noting the design flexibility imposed by the update scenario: either imposed by the intrinsic non-determinism of consistency maintenance or by side-effect ordering. Our current solution implements pre-defined choices.

Indeed, on line 1 of Algorithm 1, each condition c, necessary for applying r_U on $\mathbb{G}$, is added to *PreCoditions*. Function *FindPredCond2ApplyUpd* works on table UpdCond indexed by the update type. Fig. 12 shows an extract of UpdCond (*e.g.* from the first row, we know that the insertion of $CI(A, B)$, depends on the existence of A as an individual, B as an class and the respect of hierarchical constraints). The full table can be found in appendix D.

Roughly, to design UpdCond for an insertion P, we consider all constraints $c \in \mathscr{C}$ having atoms with the predicate of P in $body(c)$ and we build updates corresponding to the atoms in $head(c)$. Deletions are treated in a reciprocal way: we look from the predicate of P on the head of constraints and define the new updates based on the atoms in their bodies. Unfortunately, a deletion may engender non-deterministic side-effects. Consider for instance the deletion of a class instance $CI(A, B)$. Constraint 26 in $\mathscr{C}$ (Fig. 3) indicates two possible side effects in this case: deleting A as a class instance of all super-classes of B or breaking the class hierarchy. In this paper, we deal with non-determinism in an arbitrary way: when a choice is needed, the priority is given to updates on the instance, leaving the schema unchanged. Thus, in the above example, side-effect updates are: $\neg CI(A, y_c)$, for each y_c which is a super-class of B. When non-determinism is over two side-effects implying changes on the schema, the priority is to break the highest hierarchical link.

Algorithm 1: SetUp $(\mathbb{G}, \mathbb{R}, U)$

Input: Graph database $\mathbb{G}$, set of rewriting rules $\mathbb{R}$, update U
Output: New graph database $\mathbb{G}$

```
1: PreConditons := FindPredCond2ApplyUpd(𝔾, ℝ, U)
2: for all condition c in PreConditons do
3:     if c is not satisfied in 𝔾 then
4:         U' := Planner2FitGraphInCond (𝔾, c)
5:         for all update u' in U' do
6:             𝔾 := SetUp (𝔾, ℝ, u')
7: 𝔾 := GraphRewriter(𝔾, ℝ, U)
8: return 𝔾
```

Then, on line 2 of Algorithm 1, each condition c is considered. The order in which each c is treated impacts the order in which new updates are applied to the database and gives rise to different approaches. *PreConditions* can be seen as a set (updates treated on any order) or as a list ordered according to a particular method. Our prototype uses an arbitrarily pre-defined order. Arbitrary choices

are seldom the best solution: we are currently working on a cost function to guide our choices both for update ordering and non-deterministic choices.

Once a condition c is chosen, function *Planner2FitGraphInCond* (line 4) generates a new update U' (*i.e.*, a side effect of U). Recursive calls (line 6) ensure that each side effect U' is treated in the same way. When conditions for a rewriting rule $r_{U'}$ hold, function *GraphRewriter* applies $r_{U'}$ and the graph evolves. Eventually, if U is not intrinsically inconsistent, we obtain a new graph on which r_U is applicable. Indeed, some updates U related to a fact F may be intrinsically inconsistent, *i.e.*, $\forall \mathscr{D}, F \in \mathscr{D} \implies \neg(\mathscr{D}, \mathscr{C})$. The following example illustrates such a situation.

Example 3. Let U be an intrinsically inconsistent update requiring the insertion of a class instance $CI(Excipient, Excipient)$ in $\mathbb{G}$ of Fig 1 (rule r_{CI}). Following Algorithm 1 and the order established in table UpdCond, conditions $c1 : Ind(Excipient)$ and $c2 : Cl(Excipient)$ are obtained by *FindPredCond2Apply-Upd*. However, these two conditions are contradictory since they engender inconsistent update requests, namely: $Ind(Excipient)$ and $\neg Ind(Excipient)$ and also $Cl(Excipient)$ and $\neg Cl(Excipient)$. In this situation, our current implementation behaves as follows:

1. As condition $c1$ is not satisfied by $\mathbb{G}$, insertion $Ind(Excipient)$ is required (rule r_{ind}). Rule r_{ind} imposes the deletion $\neg Cl(Excipient)$ (since *Excipient*, as an individual, cannot be a class). The deletion is performed with success, r_{ind} applies and $Ind(Excipient)$ is inserted in $\mathbb{G}$.
2. Condition c_2 is then checked. As *Excipient* is no more a class, the insertion of $Cl(Excipient)$ is triggered (rule r_{Cl}). To apply r_{Cl}, the deletion $\neg Ind(Excipient)$ is executed. Class node $Cl(Excipient)$ is added to $\mathbb{G}$.
3. Conditions c_1 and c_2 having been handled, r_{CI} is invoked but it cannot be applied: there is no individual node *Excipient*; the algorithm stops.

□

According to the method chosen for dealing and ordering side-effects on line 2 of Algorithm 1, inconsistent updates may result in cycles. As shown in Example 3, the current version of SetUp performs updnglingates according to a pre-established order, without any backtracking. Therefore, once a rule is activated for side effect e_1 of update u_1 it will not be activated again for the same update u_1. Being simple it avoids loops in the treatment of intrinsically inconsistent updates. The resulting graph, in this case, does not change, but stays consistent.

The goal of side-effects is to adapt the knowledge graph to the application of rule r corresponding to a given update U. If r is not applicable to $\mathbb{G}$ then we have: (I) $\mathbb{G}^1 = r_1(\mathbb{G})$, $\mathbb{G}^2 = r_2(\mathbb{G}^1)$, $\ldots$ $\mathbb{G}^n = r_n(\mathbb{G}^{n-1})$ where $r_1, r_2, \ldots r_n$ are the rewriting rules associated to updates recursively generated by Algorithm 1 and (II) $\mathbb{G}' = r(\mathbb{G}^n)$ is the new updated graph.

Lemma 2 (SetUp Correction and terminaison). *Let U be an update, $\mathbb{G}$ a consistent graph, $\mathbb{R}$ our set of graph rewrite rules and $\mathbb{G}' = SetUp(\mathbb{G}, \mathbb{R}, U)$. (1) SetUp terminates and (2) if U is consistent then if U is an insertion, U appears in $\mathbb{G}'$, else U is a deletion, U does not appear in $\mathbb{G}'$.*

Proof. Proof is made for each possible update. So in all cases, we show that we have (i) a finite number of side effects produced (possibly recursively) as well as (ii) the side effects corresponding to the desired update if it is consistent. All proofs are provided in Appendix C. The number of side effects is less important than the number of modifications (addition / deletion of a node or addition / deletion of an arc) in the graph because, in the SPO approach, dangling edges are automaticaly deleted, if we delete node with $Uri = A$, all edges (A, X) or (X, A) no longer exist.

6 Implementation

SetUp is implemented using Java and AGG (The Attributed Graph Grammar System) [19]. AGG is one of the most mature and cited development environment supporting the definition of typed graph rewriting systems [18]. It supports the SPO approach as well as its main extension: PAC, NAC, and GAC. The current version of *SetUp* [3] provides a textual interface and offers different updating modes, according to the user level. The complexity of *GraphRewriter* essentially determines SetUp's complexity.

Fig. 13 illustrates the recursive execution of SetUp for the insertion of the fact $PI(Aspirin, Fever^{-}, Produces)$ on $\mathbb{G}$ of Fig. 1. Leaves represent updates that are already implemented in $\mathbb{G}$ or have no more consequences. Indeed, to apply U, our algorithm generates a set S of updates that would ensure the necessary conditions for the application of the rewriting rule r_U.

Each update $u \in S$ corresponds to a rewriting rule r_u. However, when u implies no graph transformations (*e.g.* insertion of a node which already exists; deletion of an edge which does not exist, etc.) rule r_u is not executed. This is illustrated in Fig. 13 by the insertion of $Pr(Produces)$: a node concerning this property already exist in $\mathbb{G}$. On the other hand, the rule corresponding to the insertion of $Aspirin$ as a class instance of $Mollecule$ cannot be applied before the perfomance of other updates, namely: $Ind(Aspirin)$, $Cl(Molecule)$, $CI(Aspirin, rdfs : Resource)$, $CI(Aspirin, Drug)$ and $CI(Aspirin, Component)$. The execution of each update u follows the same reasoning, resulting in a graph on which r_U is executed.

The execution tree supports our approach as our algorithm correctly applies some required updates in order to get a graph $\mathbb{G}$ after the update, that satisfy all constraints. Experiment evaluations are discussed in the next section.

7 Experimental evaluation

SetUp is implemented in Java and relies on AGG (The Attributed Graph Grammar System) [19], one of the most mature and cited development environment supporting the definition and application of typed graph rewriting systems [18]. The current version of SetUp –available online [3]– provides a textual interface and offers different updating modes, according to the user level. This section

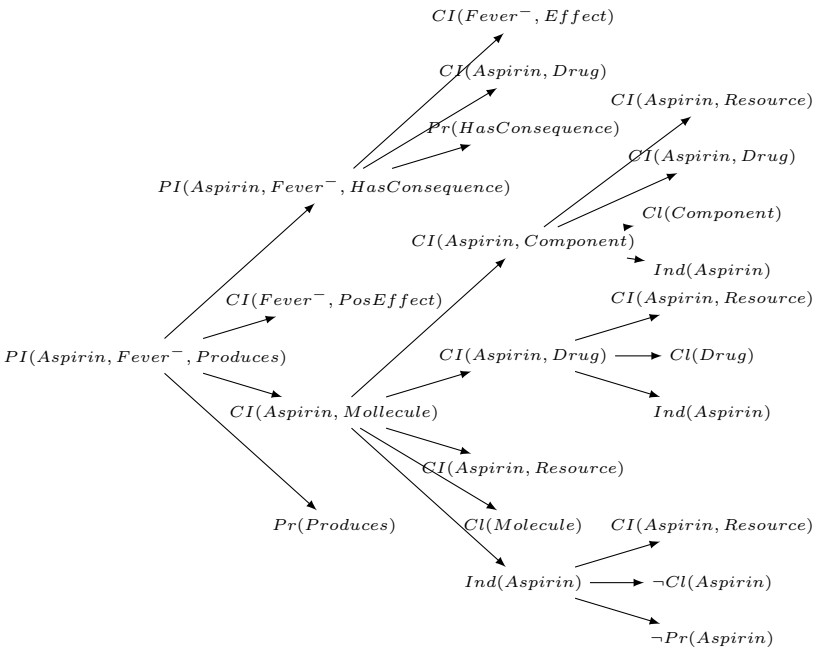


Fig. 13. Recursive execution tree

experimentally investigates the number of side-effects generated by SetUp and the time required to apply both side-effects and the original update, regarding various update scenarios.

Datasets. The experiments are conducted on synthesised RDF/S graphs that are composed of: (A) A simple hierarchy of S classes and properties where a sub-class (resp. sub-property) relation exists between each couple of classes (resp. property). (B) Instances of these classes and properties. The bottom concept in the hierarchy has I instances, the next has $2 * I$ instances, etc (so that the top of the hierarchy has S^*I instances. Fig. 14 provides an example graph with the aforementioned graphical representation.

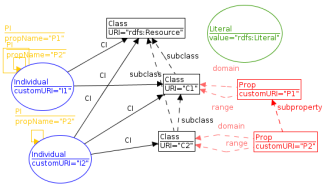


Fig. 14. Experimental graph with $I = 1$ and $S = 2$

The values of (I, S) used in experiments are $(1, 1)$, $(1, 5)$, $(5, 1)$, and $(5, 5)$ which correspond to graphs with $(|V|, |E|)$ equal to $(16, 24)$, $(44, 152)$, $(40, 80)$, and $(116, 480)$, respectively.

Experimental scenarios. Experiments consist in facts insertions and deletions as summarized in Table 1. They are categorized according to the update type and the database configurations, since the impact of an update is intrinsically related to these two factors. Every case having a check-mark indicates a scenario taken into account in our experiments, for the referenced update. As an example,

consider the insertion of an instance of class C. If C is not yet in the base but a property P with the same URI is, then P (and all its instances) are deleted to allow C's insertion. Different scenarios are defined according to the position of P in the hierarchy (lines with $\exists P_{=C}$ and $\exists P_{=CinH}$).

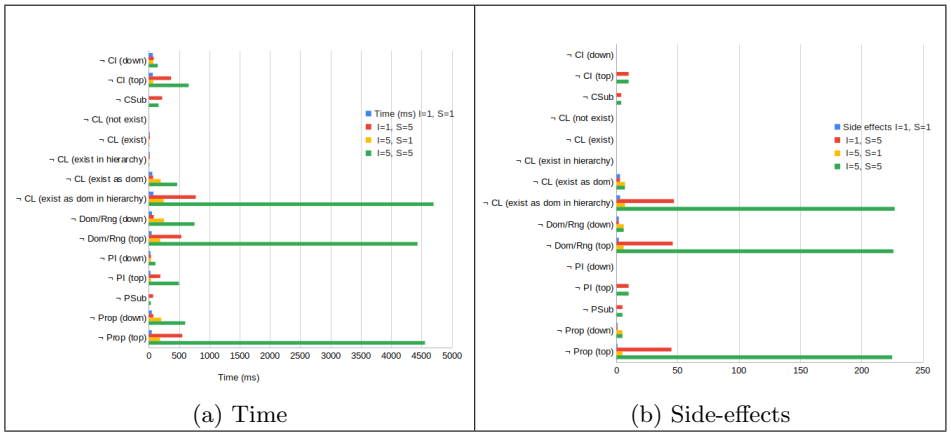
Table 1. Experiment scenarios (C is a class, P a property and I an individual.)

Scenario		*Update type*										
Notation	*Explanation*	$\neg CI$	$\neg CL$	$\neg CSub$	$\neg Prop$	$\neg PSub$	$\neg Dom$	CI	CL	$CSub$	$Prop$	$PSub$
down	Update at the bottom of the hierarchy *e.g.* $CI(Nausea, NegEffect)$	✓	✓	✓	✓	✓	✓	✓		✓		✓
top	Update at the top of the hierarchy *e.g.* $\neg CL(Effect)$	✓	✓		✓		✓	✓		✓		✓
down reverse	Update on top of the hierarchy's bottom *e.g.* $CSub(NegEffect, HealthThreat)$									✓		✓
top reverse	Update on top of hierarchy's top *e.g.* $PSub(HasConsequence, AssociatedWith)$									✓		✓
$\neg\exists C$	C is absent from the database		✓					✓	✓		✓	
$\exists C$	C exists outside any hierarchy		✓					✓				
$\exists C_{inH}$	C is at the bottom of the hierarchy (is at the top for deletion)		✓					✓				
$\exists C_{asDom}$	C is the domain of some property		✓					✓				
$\exists C_{topDom}$	C is the domain of the property and it is at the top of the hierarchy		✓					✓				
$\exists I$	the individual is already in the database							✓				
$\neg\exists I$	the individual is not in the database							✓				
$\exists P_{=C}$	there exist P with the URI of C P is outside an hierarchy							✓				
$\exists P_{=CinH}$	there exist P with the URI of C P is at the top of a hierarchy							✓				

Experimental results. Figs. 15 and 16 show the results of our experiments for deletions and insertions, respectively. The time is measured with JMH [**?**] on 3 forks of 10 warmups and 50 measure iterations with a mean of 150 op./iteration.

Side-effects tackled by the *GraphRewriter* are not taken into account: for instance, the deletion of a class at the top of the hierarchy is reported with 0 side-effect since deletion of relevant CI and $CSub$ are handled by the *GraphRewriter* through the removal of dangling edges. On the contrary, those generated by SetUp are counted even if they do not need to be applied due to the database configuration. For instance the insertion $CL(A)$ has two side effects ($\neg Pr(A)$ and $\neg Ind(A)$) that are included even if the original database does contain neither such a property nor such an individual. The number of generated side-effects varies according to the update type and the scenario. For instance, $CL(A)$ always

generates the 2 aforementioned side-effects. As $CSub$ and CI relationships are suppressed by the *GraphRewriter*), update $\neg CL(C)$ generates 0 side-effects in scenarios $\neg\exists C$, $\exists C$, and $\exists C_{inH}$. However, in the scenario $\neg Dom/Rng$ top, 2, 46, 6, and 226 are generated with $(I, S) = (1, 1)$, $(1, 5)$, $(5, 1)$, and $(5, 5)$, respectively. The first generated side-effect is $\neg Pr(P)$ with P the property whose domain or range is suppressed. It itself generates I $\neg PI$ (one per instance of P) that need to be enforced beforehand. In turn, each $\neg PI$ triggers the suppression of instances of $P's$ sub-properties with the same owner and value, hence the important increase of generated side-effects with S.

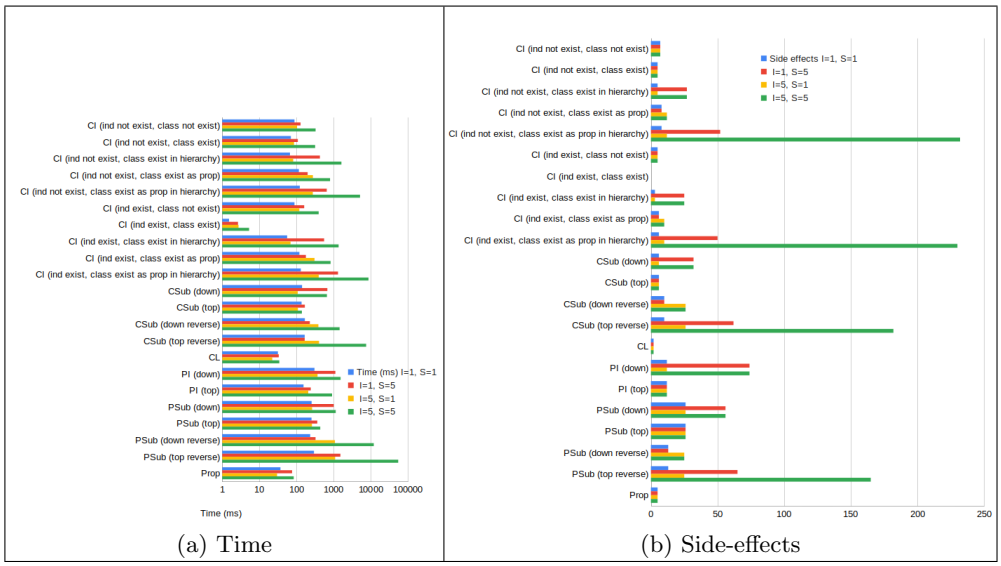


(a) Time (b) Side-effects

Fig. 15. Experimental results for fact suppression.

Execution time grows with the size of the knowledge graph, as it impacts the pattern matching and the verification of rule applicability phases. The scale of this impact varies depending on the complexity of the applied graph rewriting rules. $CL(A)$, for example, triggers the same number of side-effects by both SetUp and the *GraphRewriter* $(1, CSub(A, \text{“}rdfs : Resource\text{”})$ regardless of I and S. The applicability conditions of the corresponding rule are quite simple (two NACs) and the impact is thus marginal: it takes $31,5$s and $34,4$s with $S = 1, I = 1$ and $S = 5, I = 5$, respectively. This corresponds to a 9% execution time increase for a graph containing roughly 7 times more vertices and 20 times more edges. On the contrary, consider $\neg CI(top)$ whose rule contains complex GACs. Side effects depends solely on S and, with $S = 5$ and 10 side-effects, the execution time goes up by 79% from $I = 1$ $(368,5$s) to $I = 5$ $(660,5$s). By roughly tripling the size of the graph, each update $\neg CI$ therefore takes almost 72% more time to be executed.

The second factor impacting time is the number of generated side-effects, as they triggers calls to the pattern matching and graph rewriting algorithms. For instance, configuration $(I, S) = (1, 5)$ is bigger that $(5, 1)$ (almost as many nodes



(a) Time (b) Side-effects

Fig. 16. Experimental results for fact insertion.

but twice the number of edges), yet $\neg Dom/Rng$ (top) is almost three times longer on the second (190, 5s and 538s, respectively as the number of generated side-effects goes from 6 to 46. Notably, side-effects handled by the *GraphRewriter* mildly impact execution time: $\neg Cl$, for example, has almost the same execution time with configurations $\exists C$ and $\exists C_{inH}$ (10, 0 and 10, 3s respectively with $I = S = 5$), even tough the latter implies suppression of $CSub$ relations through the *GraphRewriter*.

8 Related work

The validity of knowledge databases is yet a very important research issue, bringing the question concerning their updating in front of the scene. Several recent work consider this problem. The focus of [15] is completely different from ours: the authors neither care about the validity *w.r.t.* constraints nor propose an updating method. They propose a semantic consistency measure on the basis of the difference between a sub-graph from the original RDF graph database and a proposed updated sub-graph. The incremental update algorithm proposed in [6] has similarities with the updating proposal in [4] but without the deep examination of null values introduced in the latter. A parallel can be done between Saturation in [6], the chase in [5,4,10] and *SetUp*. Authors in [5,6,4,10] offer home-made procedures to implement their methods: [6] deals only with the instance constraints of Fig. 3; in [5,4], constraints are tuple-generating-dependencies (tgd) given by a user while in [10] constraints are simpler than tgd (but the paper also focuses

on RDF/S semantic constraints). Nulls (as incomplete information or as a blank node in RDF) are the focus of [10,4]. All these works consider updates; [5,6] mention schema evolution. The originality of *SetUp* is the use of home-made procedures *only* to generate side-effects lying on well-studied graph rewriting system to ensure database (instance or schema) consistent evolution.

To the best of our knowledge, [13] is the only proposal regarding the use of graph rewriting to model ontology updates that provides an implementation. The objective is to tackle the evolution, alignment, and merging of ontologies (see also [14]). Graph rewriting rules are used to model updates on an OWL ontology with the open world assumption (OWA) under some consistency constraints. Their work does not consider nested and general application conditions and can thus not tackle more complex constraints (*e.g.* , relative to transitive properties). This aspect, coupled with the OWA, makes it impossible to guarantee the preservation of all of the consistency constraints considered in this paper (*e.g.* , the absence of cycles in subclass relationships).

9 Concluding Remarks

This paper proposes SetUp a theoretical and applied framework for ensuring consistency when a RDF knowledge graph evolves. Its originality lies in the use of a typed graph rewriting system offering a formal graphical specification; each atomic updates is formalized using a graph rewriting rule whose application *necessarily* preserves RDF intrinsic semantic constraints. If an update can not be applied, depending an user level, SetUp generates additional consistency preserving updates to ensure its applicability.

SetUp can be used as a test-bed for updating approaches. While its computation complexity makes SetUp unfit for on-the-fly automated updates, it is satisfactory for interactive command line updates and can also be used for offline modifications. In particular, we plan on using SetUp to work on offline RDF knowledge graph anonymization where a snapshot of a graph is updated to be anonymized and published.

References

1. Abiteboul, S., Manolescu, I., Rigaux, P., Rousset, M.C., Senellart, P.: Web data management. Cambridge University Press (2011)
2. Chabin, J., Eichler, C., Halfed Ferrari, M., Hiot, N.: Graph rewriting rules for consistent RDF updates, {Online,}https://tinyurl.com/rsyrkkr
3. Chabin, J., Eichler, C., Halfed Ferrari, M., Hiot, N.: SetUp: a tool for consistent updates of rdf knowledge graphs, {Online,}https://tinyurl.com/qwfdgwg
4. Chabin, J., Halfeld Ferrari, M., Laurent, D.: Consistent updating of databases with marked nulls. Knowledge and Information Systems (2019)
5. Flouris, G., Konstantinidis, G., Antoniou, G., Christophides, V.: Formal foundations for RDF/S KB evolution. Knowl. Inf. Syst. **35**(1), 153–191 (2013)
6. Goasdoué, F., Manolescu, I., Roatiş, A.: Efficient query answering against dynamic rdf databases. In: Proceedings of the 16th International Conference on Extending Database Technology. pp. 299–310. ACM (2013)
7. Golas, U., Biermann, E., Ehrig, H., Ermel, C.: A visual interpreter semantics for statecharts based on amalgamated graph transformation. ECEASST **39** (01 2011)
8. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inf. **26**(3,4), 287–313 (Dec 1996), http://dl.acm.org/citation.cfm?id=2379538.2379542
9. Habel, A., Pennemann, K.h.: Correctness of high-level transformation systems relative to nested conditions. Mathematical. Structures in Comp. Sci. **19**(2), 245–296 (Apr 2009)
10. Halfeld Ferrari, M., Hara, C.S., Uber, F.R.: RDF updates with constraints. In: Knowledge Engineering and Semantic Web - 8th International Conference, KESW, Szczecin, Poland, Proceedings. pp. 229–245 (2017)
11. Halfeld Ferrari, M., Laurent, D.: Updating RDF/S databases under constraints. In: Advances in Databases and Information Systems - 21st European Conference, ADBIS, Nicosia, Cyprus, Proceedings. pp. 357–371 (2017)
12. Löe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science **109**(12), 181 – 224 (1993)
13. Mahfoudh, M.: Adaptation d'ontologies avec les grammaires de graphes typés : évolution et fusion. (Ontologies adaptation with typed graph grammars : evolution and merging). Ph.D. thesis, University of Upper Alsace, Mulhouse, France (2015)
14. Mahfoudh, M., Forestier, G., Thiry, L., Hassenforder, M.: Algebraic graph transformations for formalizing ontology changes and evolving ontologies. Knowledge-Based Systems **73**, 212 – 226 (2015). https://doi.org/https://doi.org/10.1016/j.knosys.2014.10.007, http://www.sciencedirect.com/science/article/pii/S0950705114003748
15. Maillot, P., Raimbault, T., Genest, D., Loiseau, S.: Consistency evaluation of RDF data: How data and updates are relevant. In: Tenth International Conference on Signal-Image Technology and Internet-Based Systems, SITIS 2014, Marrakech, Morocco, November 23-27, 2014. pp. 187–193 (2014)
16. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1997)
17. Runge, O., Ermel, C., Taentzer, G.: Agg 2.0 – new features for specifying and analyzing algebraic graph transformations. In: Schürr, A., Varró, D., Varró, G. (eds.) Applications of Graph Transformations with Industrial Relevance. pp. 81–88. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

18. Segura, S., Benavides, D., Ruiz-Cortés, A., Trinidad, P.: Automated Merging of Feature Models Using Graph Transformations, pp. 489–505. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
19. Taentzer, G.: Agg: A graph transformation environment for modeling and validation of software. In: AGTIVE (2003)

A RDF/S databases as a typed graph

RDF/S databases are formalized in two ways in this paper: as classical triple-based RDF statements and as a typed graph. This appendix is dedicated to the latter.

RDF/S type graphs comprise 4 node types (Class, Individual, Literal, and Prop) and 6 edge types (CI, PI, domain, range, subclass, and subproperty). Each nodes have one attribute representing an URI, an URI, a value, and a name, respectively. PI-typed edges are the only ones with an attribute which represent the name of the property the edge is an instance of.

Fig 17 describes how each RDF triples are formalized in the typed graph model.

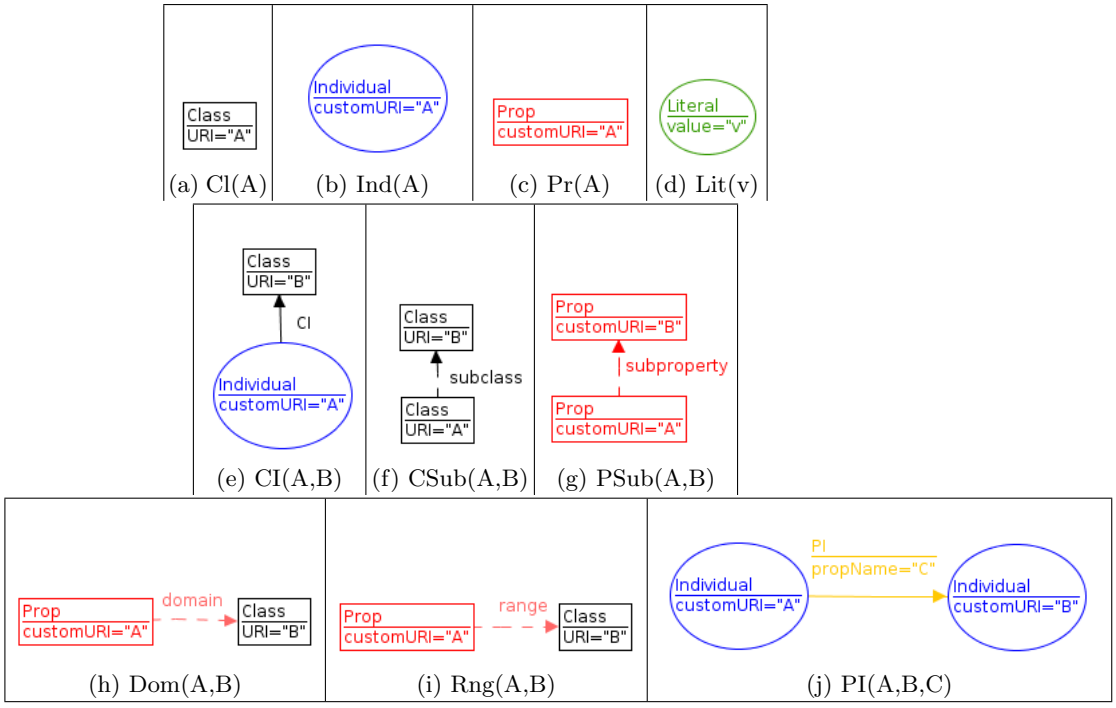


Fig. 17. RDF triples in the type graph model

B Rewriting rules formalizing consistent RDF/S updates

This section presents the graph rewriting rules which are not described in the main text, together with, for each one, the proof of its correction, *i.e.*, the proof that, when it is applied, graph consistency is preserved and the specified transformation is performed. In total, our approach works with a total of 18 graph rewriting rules.

To present the rules, in this section, we continue using the standard basic form used in Section 4. We recall that the LHS of a SPO specification is related to its applicability while its RHS is related to the effect of its application. One may see the LHS and RHS of a rule as a "before" and "after" its application; an image of the LHS has to be in the graph for the rule to be applicable and an image of the RHS will be in the graph after its application. GACs, PACs and NACs have no impact on the effect of a rule, but only restrict its applicability.

B.1 Deletion of a class instance (Fig. 18)

Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 18a)
LHS: An individual is linked to a class by an edge typed CI, *i.e.*, in the database, the individual is an instance of this given class.
RHS: The CI edge is removed (the individual still exists but is not an instance of the given class anymore).
2) NACs: The individual considered here is an instance of the given class. The NAC in Fig. 18b forbids the application of the rule when this individual is also connected to another individual by a property (*i.e.*, as part of a property instance) whose domain is the given class. The NACs in Figs. 18c and 18d are similar to the previous one, treating the cases where the individual is connected to a literal or to itself, respectively. The NACs in Figs. 18f and 18g impose similar prohibition when the given class is the range of the property. The NAC in Fig 18g ensures that no instance of resource is removed – since an individual is always an instance of class Resource. The NAC in Fig 18h disallows the rule application when the individual is an instance of a subclass of the given class.

Proof of consistency preservation: From Fig. 3, we remark that constraints 14, 24, 25, and 26 are concerned by the deletion of a class instance since an atom with predicate CI appear in their right-hand sides. The NAC in Fig 18g ensures the satisfaction of constraint 14. The NACs in Figs. 18b– 18f ensures the satisfaction of constraints 24 and 25. Finally the NAC in Fig 18h guarantees that constraint 26 is not violated.

B.2 Insertion of an individual (Fig. 19)

Update category: Instance evolution
User level: Any user

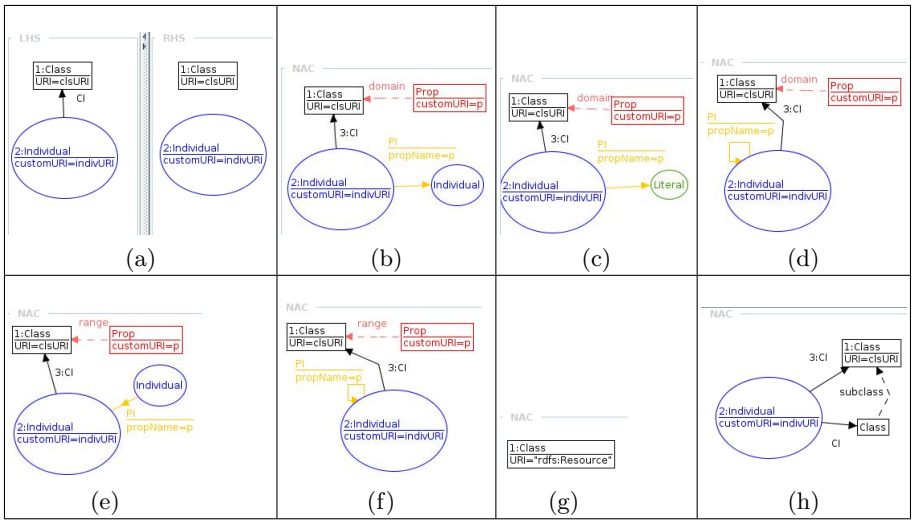


Fig. 18. Rule concerning the deletion of a class instance.

Rule semantics:
1) SPO specification: (Fig. 19a)
LHS: The class Resource;
RHS: LHS plus an individual with the URI A and a CI edge from the from the former to the latter. The application of the rule inserts the individual as an instance of class Resource.
2)NACs: The NACs defined in Fig. 19c and 19d guarantee that the sets of classes, properties, and individuals are disjoint (constraints 5 and 6 in Fig. 3). The Nac from Fig. 19b forbids the addition of the individual if an individual with the same URI already exists.

Proof of consistency preservation: The addition of an individual triggers constraints 3 (Fig. 3) requiring an URL (given as a rule parameter) and constraints 5 and 6 which are guaranteed by the two NACs. 19c and 19d. Unicity is guaranteed by NAC 19b .

B.3 Deletion of an individual (Fig. 20)

Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification:
LHS: An individual with URI A;
RHS: the empty graph: rule's application leads to the deletion of the individual with the URI A (and all edge incident to it).

Proof of consistency preservation: From Fig. 3, the deletion of an individual may impact constraints 11 and 12. These constraints are still preserved because CI

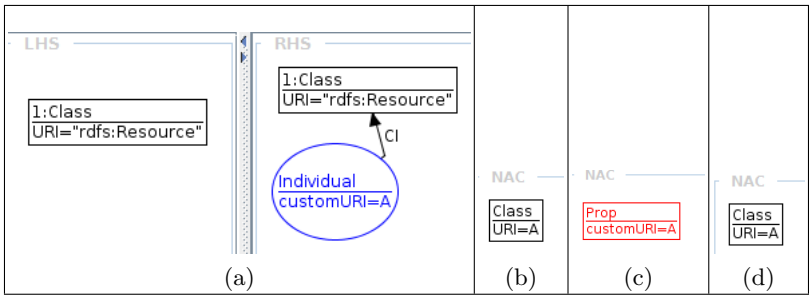


Fig. 19. Rewriting rule for the insertion of an individual

and PI relations involving an individual A are represented as an edge incident to the node modelling $Ind(A)$. In the SPO approach, dangling edges are deleted, thus all CI and PI relations involving $Ind(A)$ are suppressed when this rule is applied.

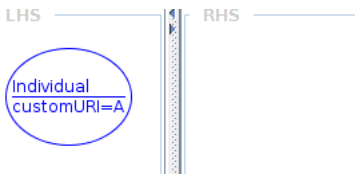


Fig. 20. Rewriting rule for the deletion of an individual

B.4 Insertion of a literal (Fig. 21)

Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 21a)
LHS: Empty
RHS: The application of the rule inserts a node corresponding to the literal and its associated value in the graph.
2)NACs: (Fig. 21b) the NAC guarantees that such a literal does not exist yet.

Proof of consistency preservation: Property or class values such as textual strings are examples of RDF literals. The addition of a literal does not trigger any constraint (Fig. 3), just allowing its future use –as a value for property for instance–. The NAC avoids literal redundancy.

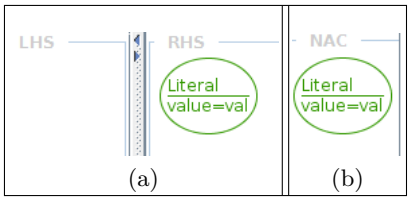


Fig. 21. Rewriting rule for the insertion of a literal

B.5 Deletion of a literal (Fig. 22)

Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 22a)
LHS: The node corresponding to the literal.
RHS: Empty.
2)NACs: (Fig. 22b) the NAC guarantees that the literal *rdfs:Literal* node is not the one been deleted; this node is a modelling artefact used as range when the range of a property is a literal and should not be deleted.

Proof of consistency preservation: From Fig. 3, the deletion of a literal is only concerned by constraint 12 when it is the value of a PI. In the SPO approach, dangling edges are deleted, thus all *PI* relations involving the literal are suppressed when this rule is applied.

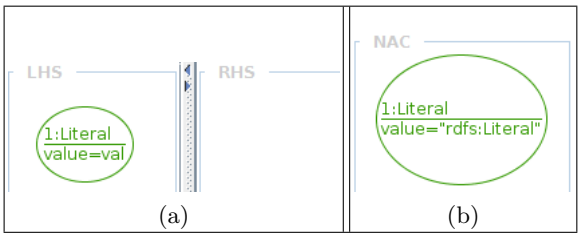


Fig. 22. Rewriting rule for the deletion of a literal

B.6 Insertion of a property

Two rules formalize the insertion of a property depending on the nature of its range.

• Insertion of a property having a class as its range (Fig. 23a)
Update category: Schema evolution

User level: Only authorized users such as database administrators
Rule semantics:
1) SPO specification: (Fig. 23a)
LHS: the LHS is composed of two classes with URI domain (denoted by *domain class*) and range (denoted by *range class*);
RHS: LHS plus a node representing a *property*, whose URI is *A*, which is connected to the domain class by a *domain*-typed edge and to the range class by a *range*-typed edge. Thus, the application of this rule inserts a property between existing classes which are specified as the domain and range of that property.
2)NACs: NACs of Figs. 24a, 24c and 24d guarantee that there exist no class with URI *A* (Fig. 24c), including the range (Fig. 24a) and the domain (Fig.24d) classes. NAC of Fig 24e prohibits the existence of an individual whose URI is *A*. Again, the NACs ensure that classes, properties, and individuals are disjoint sets (constraints 4 and 5 in Fig. 3). Finally, NAC 24b guarantees that a property with the same URI does not already exists, guaranteeing unicity.

Proof of consistency preservation: The addition of a property concerns constraints 2, 4 and 6 of Fig. 3. The NACs in Figs. 24a, 24c and 24d of our rewriting rule ensure that these three constraints are respected. Notice that classes on LHS of our rule are not required to be distinguishable. Constraint 15 in Fig. 3 is also concerned by the insertion of a property. It requires the existence of a domain and a range for every property. On the LHS, our rewriting rule imposes the existence of two classes, while in its RHS, it establishes these classes as the property's domain and range. Constraint 15 is respected even when the same class is defined as the domain and the range of a given property.

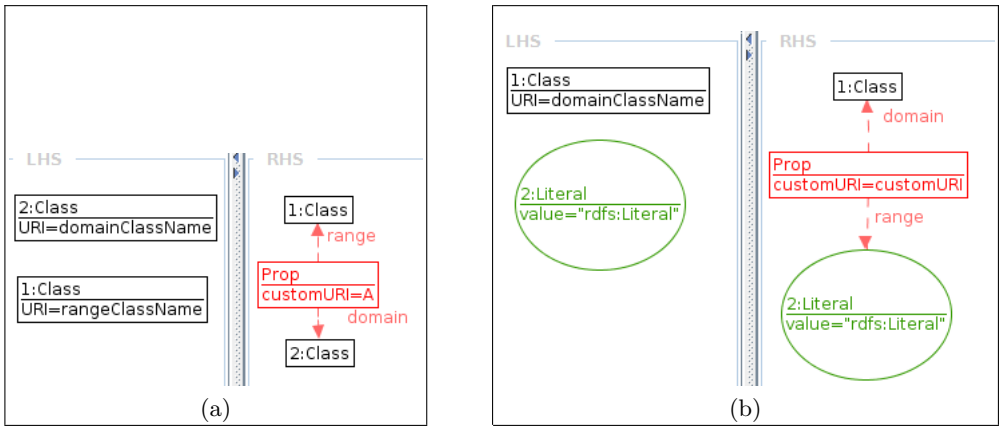


Fig. 23. Rewriting rules for inserting properties come in two versions according to the the type of the property's range.

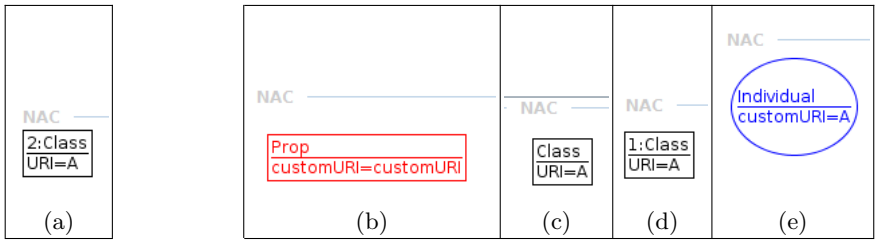


Fig. 24. NACs for the insertion of a property.

• Insertion of a property having a literal as its range (Fig. 23b)
Update category: Schema evolution
User level: Only authorized users such as database administrators
Rule semantics:
1) SPO specification: (Fig. 23b)
LHS: The LHS is composed of a class and a literal node attributed "rdfs:Literal". The latter is a special node used solely to specify that the range of a property is a literal ;
RHS: LHS plus a node representing a *property*, whose URI is A, which is connected to the class by a *domain*-typed edge and to "rdfs:Literal" by a *range*-typed edge. Thus, the application of the rule inserts a property between a class and "rdfs:Literal" which are specified, respectively, as the domain and range of that property.
2)NACs: This rule is concerned only by the four NACs defined in Fig. 24d, 24c, 24e, and 24b. These two first NACs guarantee that there exist no class with URI A (Fig. 24c), including the domain (Fig.24d) class. NAC of Fig 24e prohibits the existence of an individual whose URI is A. Again, the NACs ensure that classes, properties, and individuals are disjoint sets (constraints 4 and 5 in Fig. 3). Finally, NAC 24b guarantees that a property with the same URI does not already exists, guaranteeing unicity.

Proof of consistency preservation: The proof is similar to the previous one, the only difference is that the range is not a class, but a literal.

B.7 Deletion of a property (Fig. 25)

Update category: Schema evolution
User level: Only authorized users such as database administrators
Rule semantics:
1) SPO specification: (Fig. 25a)
LHS: a property with URI A;
RHS: the empty graph. Rule's application leads to the deletion of the property with the URI A.
2) NACs: NACs ensure that a property having instances cannot be deleted. Indeed, a property instance is a PI-typed edge between individuals (Fig. 25b

where the instances of the property are individuals), between an individual and a literal (Fig. 25c) or an atomic loop (Fig. 25d).

Proof of consistency preservation: From Fig. 3, we can remark that constraints 8, 9, 10 and 12 are concerned by the deletion of a property. Constraints 8, 9, 10 are still respected after the application of the rule because, when the node corresponding to the property is deleted, all dangling edges are deleted. Here these edges indicate sub-property relationship (constraint 8), property domain (constraint 9) or property range (constraint 10). Constraint 12 is preserved because NACs prohibit the deletion of a property having instances.

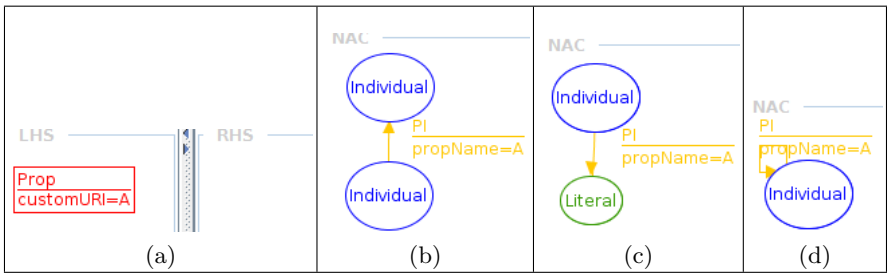


Fig. 25. Rule concerning the deletion of a property (with associated NACs).

B.8 Insertion of a property instance

We have two rules for the insertion of properties, we similarly have to consider different situations for the insertion of property instances.

• Insertion of a property instance for a property having a class as its range (Fig. 26)
Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 26a)
LHS: the LHS is composed of a property (identified by *propURI*) having classes as its domain and range. Each of these classes has an individual as an instance.
RHS: LHS plus an edge connecting the individuals. The edge represents the property instance, indicating that the individuals are related to each other by the property *propURI*.
2)GAC: (Fig. 26b) In the schema of the graph database, if the property *propURI* is a sub-property of property *supPr* (for all pattern GacTransPI), then the two individuals should be already instances of *supPr* (NestCond), *i.e.*, the rule is applied only if the individuals involved in the property instance been inserted are already related by instances of all its super-properties.
3)NACs: (Fig. 26c) The NAC guarantees that the individuals are not already instances of the property *propURI*.

Proof of consistency preservation: The addition of a property instance concerns constraints 12, 24 and 25 of Fig. 3 which are ensured by the SPO specification. Let us denote by source (respectively, target) of a PI edge the node (individual) being the start point (respectively, the ending point) of the PI edge. The LHS guarantees: (i) the existence of two individuals and the property in the graph (constraint 12), (ii) that the source of the PI is an instance of its class domain (constraint 24) and (iii) that the target of the PI is an instance of its class range (constraint 25). Constraint 27 is ensured by the GAC. An instance of P can be inserted between two individuals only if their is between the two an instance of all the super-properties of P.

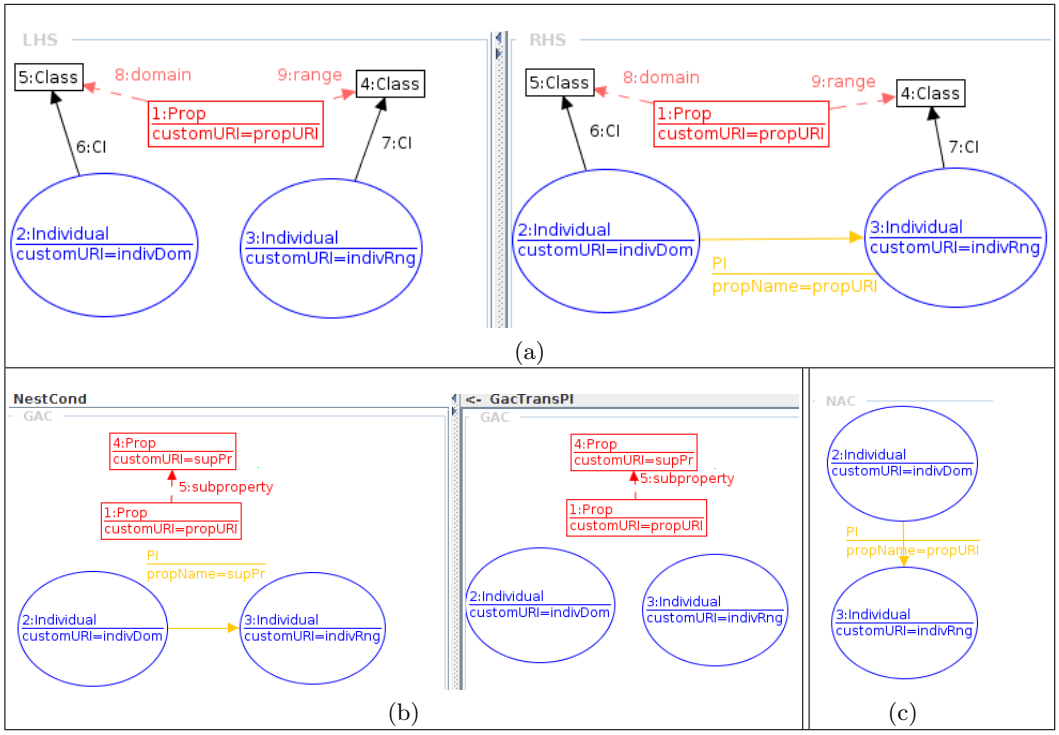


Fig. 26. Rewriting rule for the insertion of a property instance when the property range is a class.

• Insertion of a property instance for a property having a literal as its range (Fig. 27)
Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 27a)
LHS: the LHS is composed of a property (identified by *propURI*) having a class

as its domain and the literal class as its range, one individual (instance of the domain) and a literal.
RHS: LHS plus an edge from the individual to the literal. The edge represents the property instance, indicating that the individual and the literal are related to each other by the property *propURI*.
2)GAC: (Fig. 27c) In the graph database schema, if the property *propURI* is a sub-property of property *supPr* (GacTransPI), then the individual and the literal should be already instances of *supPr* (NestCond), *i.e.*, the rule is applied only if the individual and the literal involved in the property instance been inserted are already involved in instances of all its super-properties.
3)NACs: (Fig. 27b) The NAC guarantees that the individual and the literal are not already linked as an instance of the property *propURI*.

Proof of consistency preservation: Similar to the proof in the previous item.

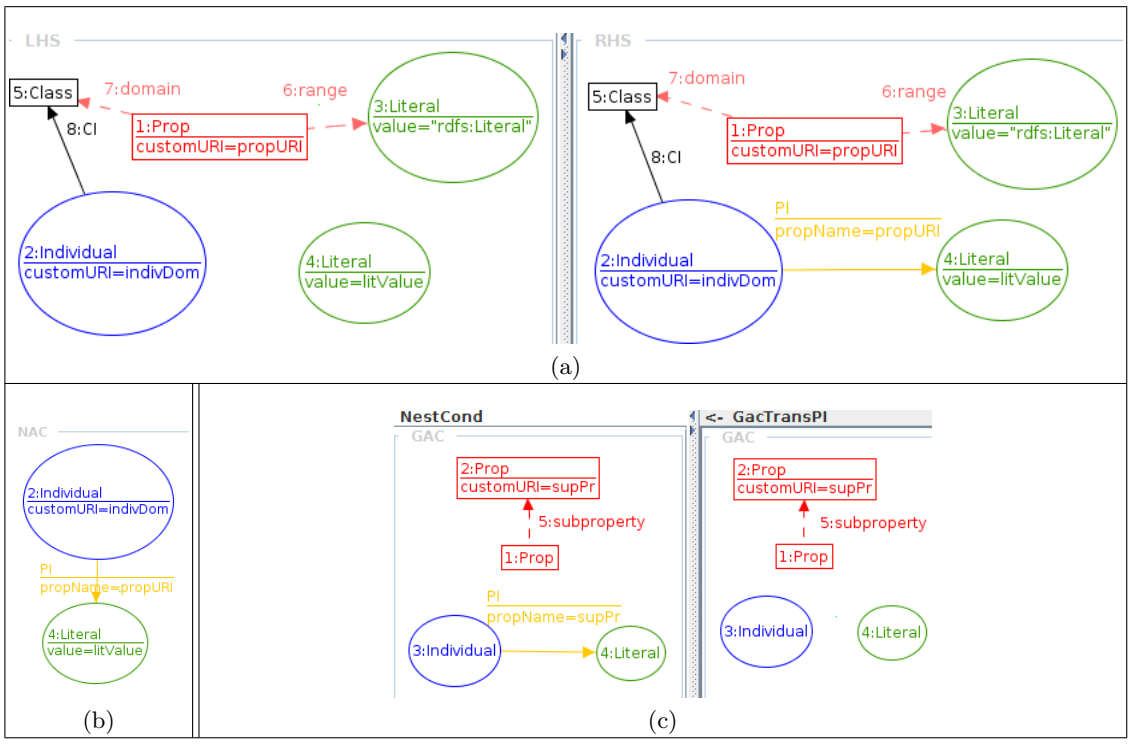


Fig. 27. Rewriting rule for the insertion of a property instance when the property range is a literal.

B.9 Deletion of a property instance

Similarly, we have to consider two different situations for the deletion of property instances.

• Deletion of a property instance for a property having a class as its range (Fig. 28)
Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 28a)
LHS: the LHS is composed of two individuals denoted by *indivDom* and *indivRng* linked by a PI-typed edge attributed with *propURI*, *i.e.*, there is an instance of property *propURI* whose object is *indivDom* and value *indivRng*.
RHS: LHS minus the edge, the rule application leads to the removal of the property instance.
2)NACs: (Fig. 28b) If property *propURI* has at least one sub-property the individuals are also instances of the NAC forbids the rule application.

Proof of consistency preservation: The deletion of a property instance concerns constraint 27 of Fig. 3. The NAC ensures this constraint since the rule cannot be triggered if there exist sub-property instance links between the individuals.

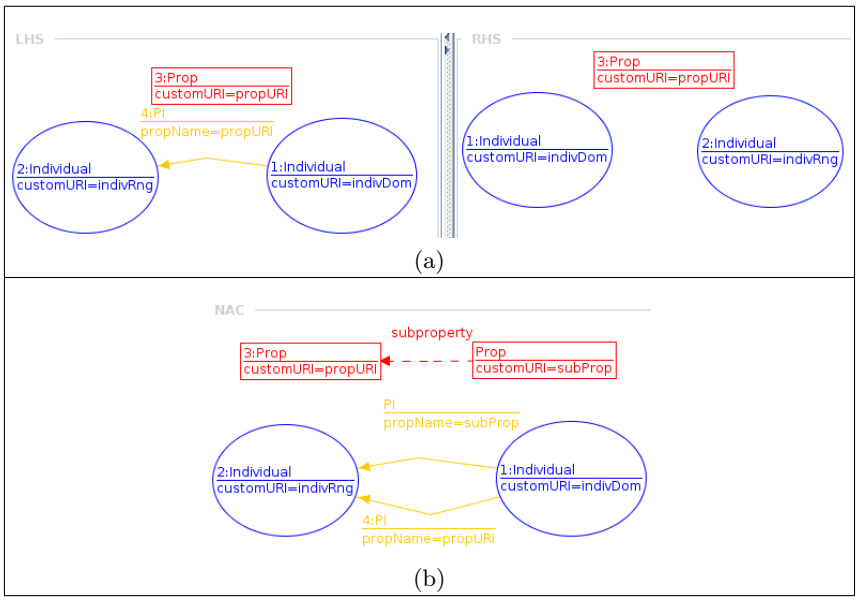


Fig. 28. Rewriting rule for the deletion of a property instance when the property range is a class.

• Deletion of a property instance for a property having a literal as its range (Fig. 29)
Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 29a)
LHS: the LHS is composed of an individuals denoted by *indivDom* and a literal linked by a PI-typed edge with attribute *propURI*, *i.e.*, they are instances of property *propURI*.
RHS: LHS minus the edge, the rule application leads to the removal of the edge linking the the individual to the literal.
2)NACs: (Fig. 29b) If property *propURI* has a sub-property with an instance involving *indivDom* and the literal, then the NAC forbids the rule application.
Proof of consistency preservation: Similar to the proof in the previous item.

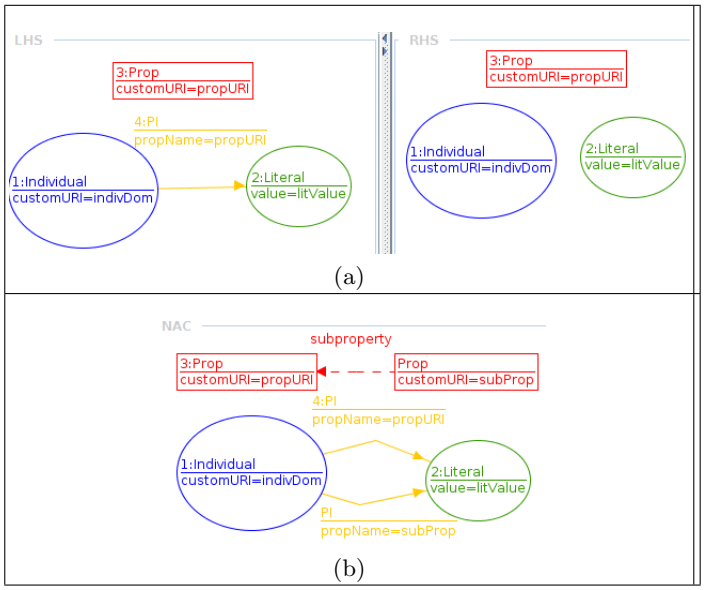


Fig. 29. Rewriting rule for the deletion of a property instance when the property range is a literal.

B.10 Insertion of a subclass relation (Figs. 30 and 31)

Update category: Schema evolution
User level: Only authorized users such as database administrators
Rule semantics:
1) SPO specification: Fig 30a

LHS: Two class-typed nodes with URI A and B;
RHS: An edge of type "subclass" from class B to class A is added, indicating that B is a subclass of A.
2) NACs: The NAC in Fig 30b ensures that class B is not a subclass of class A yet, while the NAC in Fig 30c prohibits the construction of cyclic subclass relationships – if A is already a subclass of B, the insertion of a subclass relationship from B to A is not possible. The NAC in Fig 30d forbids reflexive subclass relationship.
3) GACs: Fig. 31
The tree in Fig 31a shows the entire logical combination of conditions imposed to the graph for the application of the rule. The right branch of the tree refers to GAC in Fig. 30b. In the graph database, all individuals which are instances of class B (`GacTransCI`) should also be instances of class A (`NestCond1`) for the rule to be applicable, *i.e.*, the rule is applicable only if all instance of B are already instances of A. The left sub-tree in Fig. 31 gathers two conditions. The leftmost one corresponds to Fig 31c. Each superclass of A (`GacTransCsub`) is a superclass of B (NestCond), *i.e.*, the application of the rule is possible only if there exists a subclass relationship between B and all superclass of A. The last condition is the one in Fig.31d. It states that all subclass of class B (`GacTransCsub2`) is a subclass of A (`NestCond2`).

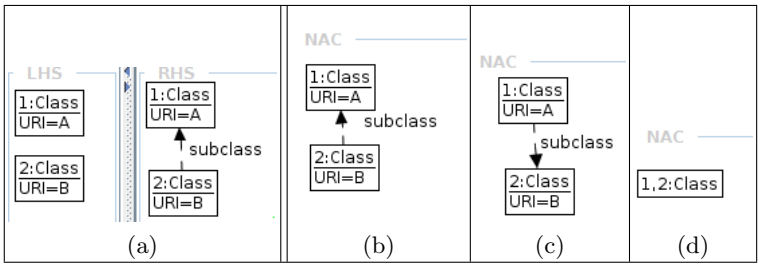


Fig. 30. Rule concerning the insertion of a subclass (with associated NACs).

Proof of consistency preservation: From Fig. 3, we remark that constraints 7, 18, 19 and 26 are concerned by the insertion of a subclass. Constraint 7 is not violated since the LHS of the SPO specification (Fig. 30a) imposes the existence of two classes before the addition of the edge representing the subclass relationship. GACs in Figs. 31c and 31d ensures the satisfaction of constraint 18 of Fig. 3, since they guarantee the application of the rule only if the class hierarchy stays consistent. Constraint 19 is implemented by the NACs which ensure that a cyclic subclass hierarchy is not possible. Constraint 26 is ensured by GAC in Fig. 31b which imposes the application of the rule only if all instances of class B are already instances of class A.

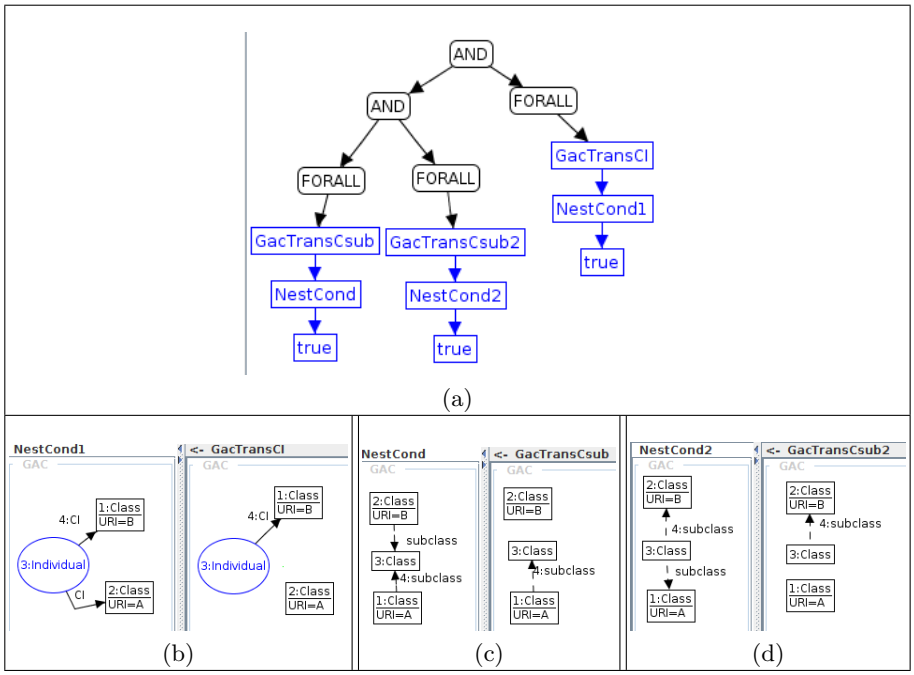


Fig. 31. GAGs concerning the insertion of a sub-class property.

B.11 Deletion of a subclass relation (Fig. 32)

Update category: Schema evolution
User level: Only authorized users such as database administrators
Rule semantics:
1) SPO specification: Fig 32a
LHS: Two class-typed nodes with URI *subClass* and *SuperClass* with a subclass-typed edge from the former to the latter;
RHS: LHS minus the edge, indicating its deletion.
2) NACs: The NAC in Fig 32b ensures that class *SuperClass* is not "rdfs:Resource", since all classes are subclasses of the root. The NAC in Fig 32c (resp. Fig 32d) ensures that *SuperClass* is not the domain (resp. the range) of a property which has a sub-property whose domain (resp. range) is *subClass*. If such properties exist, the rule is not applicable. The NAC in Fig 32e forbids the existence of a class that is both a subclass of *SuperClass* and a superclass of *subClass*, ensuring consistency with regard to transitivity.

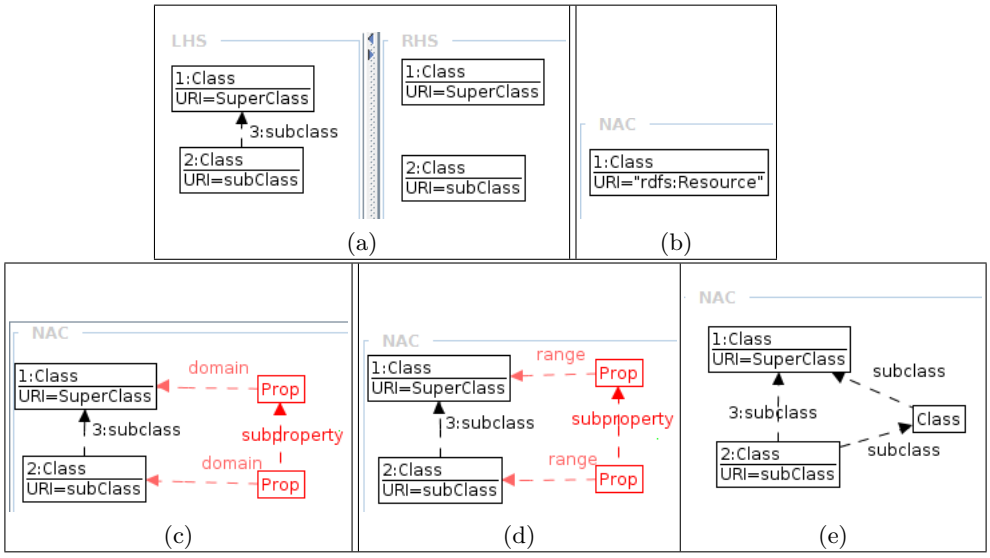


Fig. 32. Rule concerning the deletion of a subclass (with associated NACs).

Proof of consistency preservation: From Fig. 3, we remark that only constraints 13, 18, 22 and 23 are concerned by the deletion of a subclass relation. Constraint 13 is preserved thanks to the NAC defined in Fig. 32b that forbids the suppression of the sub-class relation to the root of the class hierarchy. The NAC of Fig. 32e ensures that the transitivity of the sub-class relation is respected, guaranteeing the respect of constraint 18. Finally, constraints 22 and 23 are ensured by NACs

depicted in Figs. 32c and 32d, respectively. The sub-class relationship can not be deleted if it is required for the subsumption between two properties to reflect in their domains and ranges.

B.12 Insertion of a sub-property relation (Fig. 33, 34, 35, and 36)

Update category: Schema evolution
User level: Only authorized users such as database administrators
Rule semantics:
1) SPO specification: Fig 33a
LHS: Two property-typed nodes with URI $superProp$ and $subProp$;
RHS: LHS plus a subproperty-typed edge from the node with URI $subProp$ to the one with URI $superProp$, indicating its addition.
2) NACs: The NACs in Fig 33b and 33d ensure that the sub-property relation is neither reflexive nor symmetric, respectively. The NAC formalized in Fig. 33c forbids the insertion of the relation if it already exists.
3) GACs: The logical formula for the GACs is presented in Fig 34 while the GACs are formalized in Fig. 35 and 36. The logical formula states that all of the following conditions must be fulfilled for the rule to be applicable:

- $SameDom \vee SubDom$ (Fig. 35a and 35b); the properties have the same domain or the domain of $superProp$ is a super-class of $subProp$'s domain;
- $SameRng \vee SameRngLit \vee SubRng$ (Fig. 35c), 35d, and 35e; the properties have the same range or the range of $superProp$ is a super-class of $subProp$'s domain;
- for all patterns $GacTransPI$, $NestCond$ is true (Fig. 36a); all couple of individual related with an instance of $superProp$ also have an instance of $subProp$.
- for all patterns $GacTransPISelf$, $NCselfPI$ is true (Fig. 36c); all individual with a reflexive instance of $superProp$ also has an instance of $subProp$.
- for all patterns $GacTransPILit$, $NCtransPIlit$ is true (Fig. 36b); all couple of individual and literal with an instance of $superProp$ also have an instance of $subProp$.
- for all patterns of $GacTransPsub$, $NCtransPsub$ is true (Fig.36d) (resp. $GacTransPsub2$, $NCtransub2$); all super-property of $superProp$ is also a super-property of $subProp$ (resp. all sub-property of $subProp$ is also a sub-property of $superProp$).

Proof of consistency preservation: From Fig. 3, we remark that only constraints 8, 20, 21, 27, 22, and 23 are concerned by the deletion of a subproperty relation.

The typing of the relation (constraint 8) are guaranteed by the SPO part of the rule that may match only property-typed nodes.

Constraints 21 is preserved thanks to the NACs defined in Fig. 33b and 33d that forbid the introduction of a cycle in the sub-property relation. The GACs of Fig. 36d and 36e ensure the preservation of the relation transitivity (constraint 20).

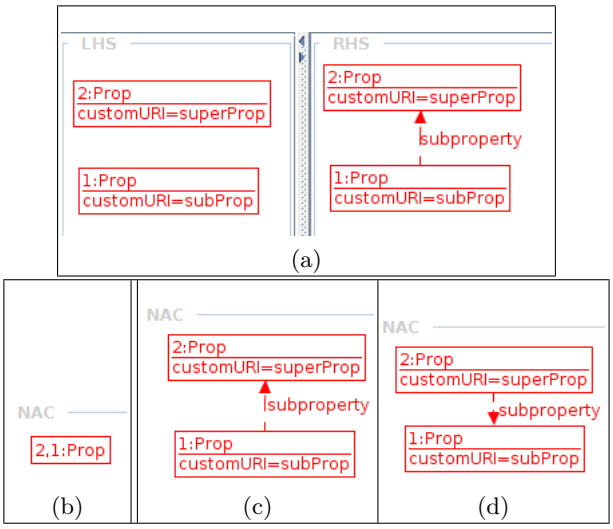


Fig. 33. Rule concerning the insertion of a subproperty relation subclass (with associated NACs).

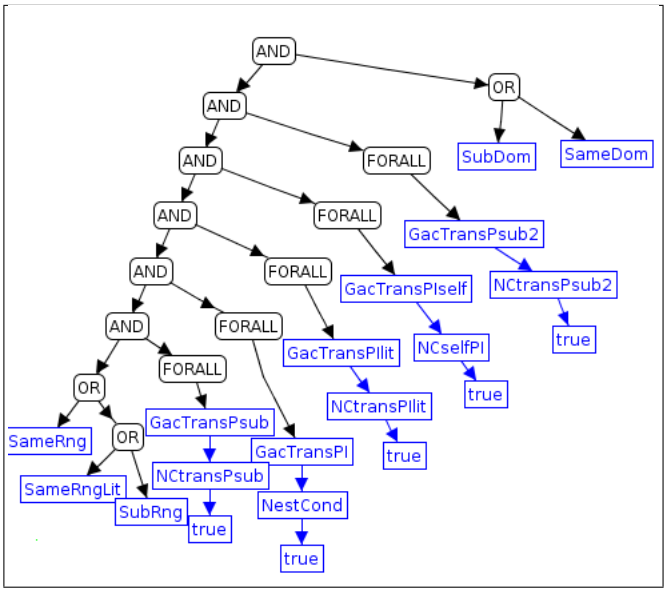


Fig. 34. Logical relations for GACs regarding the insertion of a subproperty relation subclass.

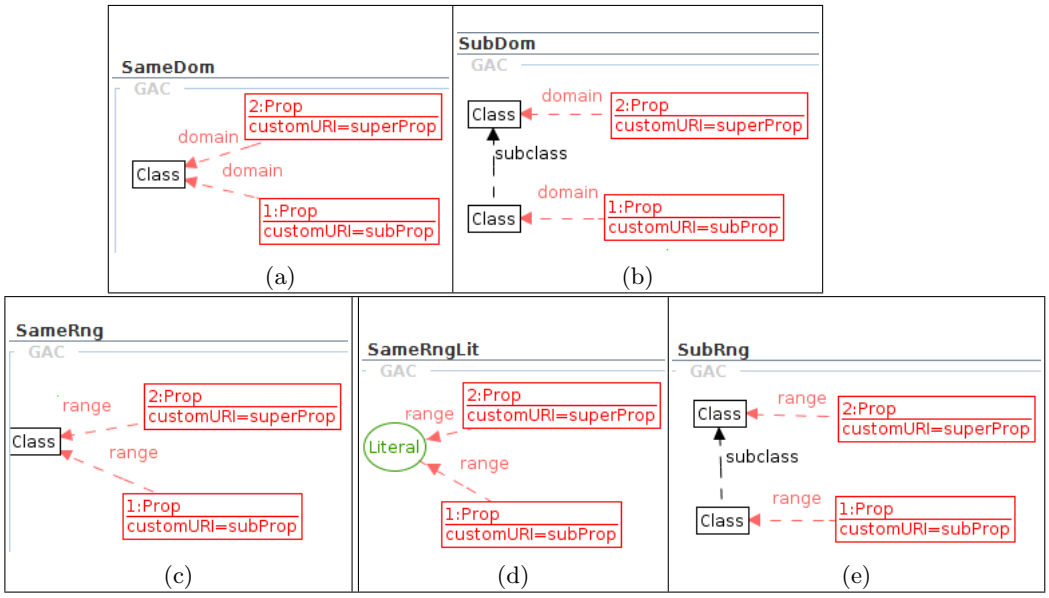


Fig. 35. GACs for the insertion of a subproperty relation subclass.

The preservation of property instance propagation (constraint 27) is ensured by the GACs represented in Fig. 36c, 36a, and 36a.

Finally, constraints 22 and 23 are ensured by GACs depicted in Figs. 35a and 35b and 35c, 35d, and 35e, respectively. The sub-property relationship may be added only if the two properties have the same domain (resp. same range) or if their respective domains (resp. ranges) are related with an adequate sub-class relationship.

B.13 Deletion of a subproperty relation (Fig. 37)

Update category: Schema evolution
User level: Only authorized users such as database administrators
Rule semantics:
1) SPO specification: Fig 37a
LHS: Two property-typed nodes with URI *superProp* and *subProp* with a subproperty-typed edge from the former to the latter;
RHS: LHS minus the edge, indicating its deletion.
2) NAC: The NAC in Fig 37b ensures that there exists no third property which is both a super-property of *subProp* and a sub-property of *superProp*.

Proof of consistency preservation: From Fig. 3, we remark that only constraint 20 is concerned by the deletion of a suproperty relation. Its conservation is ensured by the NAC of Fig. 37b that forbids deletion of the relation if it has to exist due to transitivity.

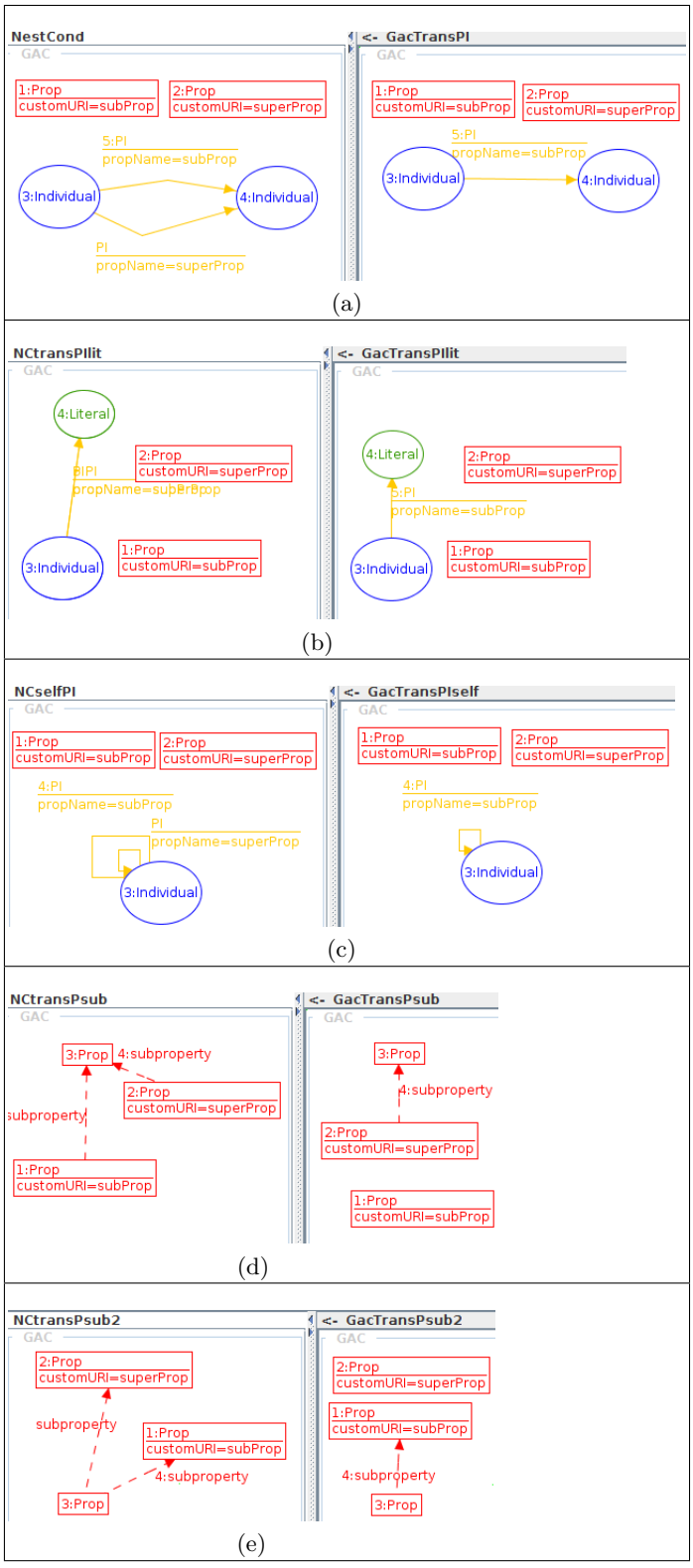


Fig. 36. GACs for the insertion of a subproperty relation subclass (cont').

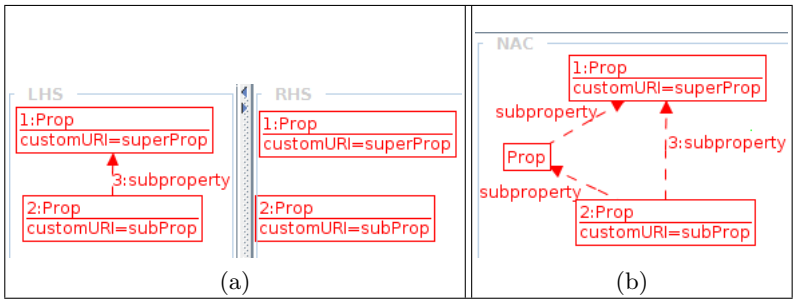


Fig. 37. Rule concerning the deletion of a subproperty relation (with associated NAC).

C SetUp proof

Lemma 3 (SetUp Correction and terminaison). *Let U be an update, $\mathbb{G}$ a consistent graph, $\mathbb{R}$ our set of graph rewrite rules and $\mathbb{G}' = SetUp(\mathbb{G}, \mathbb{R}, U)$. (1) SetUp terminates and (2) if U is consistent then if U is an insertion, U appears in $\mathbb{G}'$, else U is a deletion, U does not appear in $\mathbb{G}'$.*

Proof. Proof is made for each possible update. So in all cases, we show that we have (i) a finite number of side effects produced (possibly recursively) as well as (ii) the side effects corresponding to the desired update if it is consistent. All proofs are provided in Appendix C. The number of side effects is less important than the number of modifications (addition / deletion of a node or addition / deletion of an arc) in the graph because, in the SPO approach, dangling edges are automaticaly deleted, if we delete node with $Uri = A$, all edges (A, X) or (X, A) no longer exist.

Let study different possible updates. In the columns of the following tables, we present the side effects obtained with each recursive call from SetUp. The side effects with red background are not tested because, in the SPO approach dangling edges are automaticaly deleted, if we delete node with $Uri = A$, all edges (A, X) or (X, A) no longer exist. If a side effect does not produce any other side effect, it is either that the conditions for producing a new side effect are all false or that it is done in the rewrite rule. Side effects with green background are done in a recursively call of *SetUp*, those with yellow background are done by *GraphRewriter* line 7 of Algorithm 1 where U is the first column of the following tables.

- **For** $SetUp(\mathbb{G}, \mathbb{R}, Cl(A))$**.** (Insertion of a Class)

$Cl(A)$	$\neg Pr(A))$	$\forall X \neg PSub(X, A)$	Nothing to do
		$\forall X \neg PSub(A, X)$	Nothing to do
		$\forall X \neg Dom(A, X)$	Nothing to do
		$\forall X \neg Rng(A, X)$	Nothing to do
		$\forall X, Y \neg Pi(X, Y, A)$	No more side effects
	$\neg Indiv(A)$	$\forall X \neg CI(A, X)$	Nothing to do
		$\forall X, Y \neg Pi(A, X, Y)$	Nothing to do
		$\forall X, Y \neg Pi(X, A, Y)$	Nothing to do
	$CSub(A, Resource)$	Nothing to do, it's added by the rewwrite rule.	
	$Uri(A)$	Nothing to do, it's added by the rewwrite rule.	

With this table we conclude that $SetUp(\mathbb{G}, \mathbb{R}, Cl(A))$ ends and the result contains $Cl(A)$.

***Cl*(*A*) :**

↳ $\neg Pr(A) \rightarrow \forall X, Y \neg Pi(X, Y, A)$
↳ $\neg Indiv(A)$
↳ $CSub(A, Resource)$
↳ $Uri(A)$

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, \neg Cl(Resource))$**.** (Deletion of class Resource). We do nothing because a graph without the node $Cl(Resource))$ is inconsistent.

¬*Cl*(*Resource*)) : *nothing*

• **For** $SetUp(\mathbb{G}, \mathbb{R}, \neg Cl(A))$**.** (Deletion of a class distinct of Resource).

$\neg Cl(A)$	$\forall X \neg CSub(X, A))$	Nothing to do		
	$\forall X \neg CSub(A, X)$	Nothing to do		
	$\neg CSub(A, Resource)$	Nothing to do		
	$\forall X \neg Dom(X, A)$	$\neg Pr(X)$ *	$\forall Y, Z \neg PI(Y, Z, X)$ *	No more s-e
	$\forall X \neg Rng(X, A)$	$\neg Pr(X)$ *	$\forall Y, Z \neg PI(Y, Z, X)$ *	No more s-e

In this previous table, we introduce '*' in some cells when side effect is done for all X find in the previous cell. The number of these side effects is finite because the graph is finite.

¬*Cl*(*A*) $A \neq Resource$:

↳ $\forall X \neg Dom(X, A) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$
↳ $\forall X \neg Rng(X, A) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, CI(A, B))$**.** (Insertion of a class instance).

***CI*(*A*, *B*) :**

↳ $Cl(B)$
  ↳ $\neg Indiv(B)$
  ↳ $\neg Pr(B) \rightarrow \forall X, Y \neg Pi(X, Y, B)$
↳ $Indiv(A)$
  ↳ $\neg Cl(A)$
    ↳ $\forall X \neg Dom(X, A) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$

↳ $\forall X \neg Rng(X, A) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$
↳ $\neg Pr(A) \rightarrow \forall X, Y \neg Pi(X, Y, A)$
↳ $CI(Resource, A)$
↳ $\forall X$ s.t. $CSub(B, X)$ $CI(A, X)$

It always terminates because $\forall X$ s.t. $CSub(B, X)$ $CI(A, X)$ does't produce other side effects, indeed if it exists X, X is a node of a consistent graph so $CI(X, A)$ just puts an edge between X and $Indiv(A)$ ($Indiv(A)$ is added in the graph just before, nothing to check here). The case where we can't apply the rewrite rule is explain previously so if $A! = B$, $SetUp(\mathbb{G}, \mathbb{R}, CI(A, B))$ adds in the graph $CI(A, B)$ and all side effects to keep it consistent.

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, \neg CI(A, Resource))$**.** (Deletion of a Resource instance).
$\neg CI(A, Resource)$

↳ $\neg Indiv(A)$
↳ $\neg Literal(A)$ except if $A =' Literal'$

It terminates of course. It's correct, if we don't have $Indiv(A)$ or $Literal(A)$, the edge $CI(A, Resource)$ is dangling and so automaticaly deleted. If A is $'Litteral'$ we do nothing because a graph without the node $Literal(Literal)$) is inconsistent.

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, \neg CI(A, B))$**.** (Deletion of a class instance class is distinct of Resource).
$\neg CI(A, B)$:

↳ $\forall X s.t. Dom(X, B)$ then $\forall Y \neg PI(A, Y, X)$
↳ $\forall X s.t. Rng(X, B)$ then $\forall Y \neg PI(Y, A, X)$
↳ $\forall X s.t. CSub(X, B)$ then $\neg CI(A, X)$
  ↳ $\forall Y s.t. Dom(X, Y)$ then $\forall Z \neg PI(A, Z, Y)$
  ↳ $\forall Y s.t. Rng(X, Y)$ then $\forall Z \neg PI(Z, A, Y)$

There are no other recusive calls due to transitivity of $CSub$. It terminates because graph is finite.

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, Indiv(A))$**.** (Insertion of an individual).
$Indiv(A)$:

↳ $\neg Cl(A)$
  ↳ $\forall X \neg Dom(X, A) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$
  ↳ $\forall X \neg Rng(X, A) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$
↳ $\neg Pr(A) \rightarrow \forall X, Y \neg Pi(X, Y, A)$
↳ $CI(A, Resource)$

It terminates and $Indiv(A)$ is in the graph.

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, \neg Indiv(A))$**.** (Deletion of an individual).

$\boldsymbol{\neg Indiv(A)}$ **:** no side effects is produce all dangling edges are deleted. Terminates and $Indiv(A)$ is no more in the result graph.

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, Literal(A))$**.** (Insertion of a literal).

$\boldsymbol{Literal(A)}$ **:** just added in the graph.

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, \neg Literal(A))$**.** (Deletion of a literal).

$\boldsymbol{Literal(A)}$ **:** just removed from the graph, all dangling edge are removed.

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, Pr(A, B, C))$**.** (Insertion of a property A with it's domain B and it's range C which is $Literal$ here.)

$\boldsymbol{Pr(A, B, C)}$ **:** if $Pr(A), Dom(A, B), Rng(A, C)$ are all in $\mathbb{G}$ do nothing else :

↳ $\neg Pr(A) \rightarrow \forall X, Y \neg Pi(X, Y, A)$
↳ $\neg Cl(A)$
  ↳ $\forall X \neg Dom(X, A) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$
  ↳ $\forall X \neg Rng(X, A) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$
↳ $\neg Indiv(A)$
↳ $Cl(B)$
  ↳ $\neg Pr(B) \rightarrow \forall X, Y \neg Pi(X, Y, B)$
  ↳ $\neg Indiv(B)$
  ↳ $CSub(B, Resource)$
  ↳ $Uri(B)$
↳ if $C = Literal$ do nothing
  else $Cl(C)$:
  ↳ $\neg Pr(C) \rightarrow \forall X, Y \neg Pi(X, Y, C)$
  ↳ $\neg Indiv(C)$
  ↳ $CSub(C, Resource)$
  ↳ $Uri(C)$
↳ $Pr(A)$
↳ $Dom(A, B)$
↳ $Rng(A, C)$

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, \neg Pr(A))$**.** (Deletion of a property.)

$\boldsymbol{\neg Pr(A)}$ **:**

↳ $\forall X, Y \neg Pi(X, Y, A)$

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, Pi(A, B, C))$**.** (Insertion of a property instance.)

$\boldsymbol{Pi(A, B, C)}$ **:**

↳ if $\exists DC, RC$ s.t. $\{Pr(C), Dom(C, DC), Rng(C, RC)\} \subseteq \mathbb{G}$ then do nothing
else (do the update $Pr(C, Resource, Resource)$):
- ↳ $\neg Cl(C)$
  - ↳ $\forall X \neg Dom(X, C) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$
  - ↳ $\forall X \neg Rng(X, C) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$
- ↳ $\neg Indiv(C)$
- ↳ $Pr(C)$
- ↳ $Dom(C, Resource)$, (note $DC = Resource$)
- ↳ if $Literal(B)$ then $Rng(C, Literal)$, (note $RC = Literal$)
  else $Rng(C, Resource)$, (note $RC = Resource$)

↳ $Ci(A, DC)$
- ↳ $Indiv(A)$
  - ↳ $\neg Cl(A)$
    - ↳ $\forall X \neg Dom(X, A) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$
    - ↳ $\forall X \neg Rng(X, A) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$
  - ↳ $\neg Pr(A) \rightarrow \forall X, Y \neg Pi(X, Y, A)$
  - ↳ $CI(Resource, A)$
- ↳ $\forall X$ s.t. $CSub(DC, X)$ $CI(A, X)$

↳ if $RC = Literal$ then $Literal(B)$
else $Ci(B, RC)$
- ↳ $Indiv(B)$
  - ↳ $\neg Cl(B)$
    - ↳ $\forall X \neg Dom(X, B) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$
    - ↳ $\forall X \neg Rng(X, B) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y, Z \neg PI(Y, Z, X)(*)$
  - ↳ $\neg Pr(B) \rightarrow \forall X, Y \neg Pi(X, Y, B)$
  - ↳ $CI(Resource, B)$
- ↳ $\forall X$ s.t. $CSub(RC, X)$ $CI(B, X)$

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, \neg Pi(A, B, C))$**.** (Deletion of a property instance.)

$\neg Pi(A, B, C)$:

↳ $\forall X$ s.t. $PSub(X, C)$ $\neg Pi(A, B, X)$

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, CSub(A, B))$**.** (Insertion of a sub-class relation.) $A \neq B$ and $A \neq Resource$ else we do nothing since $CSub(A, A)$ and $CSub(Resource, B)$ are inconsistents.

$CSub(A, B)$:

↳ $\neg CSub(B, A)$
- ↳ $\forall X, Y$ s.t. $Dom(X, B)$ and $Dom(Y, A)$: $\neg PSub(X, Y) \rightarrow$
  $\forall U$ s.t. $PSub(X, U)$ and $PSub(U, Y)$: $\neg PSub(U, Y)$ (*)
- ↳ $\forall X, Y$ s.t. $Rng(X, B)$ and $Rng(Y, A)$: $\neg PSub(X, Y) \rightarrow$
  $\forall U$ s.t. $PSub(X, U)$ and $PSub(U, Y)$: $\neg PSub(U, Y)$ (*)

↳ $\forall X$ s.t. $CSub(B,X)$ and $CSub(X,A)$: $\neg CSub(X,A) \rightarrow$
$\forall Y,Z$ s.t. $Dom(Y,X)$ and $Dom(Z,A)$: $\neg PSub(Y,Z)$ (*) $\rightarrow$
$\forall U$ s.t. $PSub(Y,U)$ and $PSub(U,Z)$: $\neg PSub(U,Z)$ (**)

↳ $\forall X$ s.t. $CSub(B,X)$ and $CSub(X,A)$: $\neg CSub(X,A) \rightarrow$
$\forall Y,Z$ s.t. $Rng(Y,X)$ and $Rng(Z,A)$: $\neg PSub(Y,Z)$ (*) $\rightarrow$
$\forall U$ s.t. $PSub(Y,U)$ and $PSub(U,Z)$: $\neg PSub(U,Z)$ (**)

↳ $Cl(A)$
- ↳ $\neg Pr(A) \rightarrow \forall X,Y \neg Pi(X,Y,A)$
- ↳ $\neg Indiv(A)$
- ↳ $CSub(A, Resource)$
- ↳ $Uri(A)$

↳ $Cl(B)$
- ↳ $\neg Pr(B) \rightarrow \forall X,Y \neg Pi(X,Y,B)$
- ↳ $\neg Indiv(B)$
- ↳ $CSub(B, Resource)$
- ↳ $Uri(B)$

↳ $\forall Z$ s.t. $Ci(Z,A)$: $Ci(Z,B)$

↳ $\forall X$ s.t. $CSub(B,X)$: $CSub(A,X) \rightarrow \forall Z$ s.t. $Ci(Z,A)$: $Ci(Z,X)$ (*)

↳ $\forall X$ s.t. $CSub(X,A)$: $CSub(X,B) \rightarrow \forall Z$ s.t. $Ci(Z,X)$: $Ci(Z,B)$ (*)

↳ $\forall X,Y$ s.t. $CSub(B,X)$ and $CSub(Y,A)$: $CSub(Y,X) \rightarrow$
$\forall Z$ s.t. $Ci(Z,Y)$: $Ci(Z,Y)$ (*)

Proof ok for this case.

• **For** $SetUp(\mathbb{G},\mathbb{R},\neg CSub(A,B))$**.** (Deletion of a sub-class relation.) If $B = Resource$ or $A = B$ we do nothing.

$\neg CSub(A,B)$ ($A \neq B$ and $B \neq Resource$) :

↳ $\forall X,Y$ s.t. $Dom(X,A)$ and $Dom(Y,B)$: $\neg PSub(X,Y) \rightarrow$
$\forall U$ s.t. $PSub(X,U)$ and $PSub(U,Y)$: $\neg PSub(U,Y)$ (*)

↳ $\forall X,Y$ s.t. $Rng(X,A)$ and $Rng(Y,B)$: $\neg PSub(X,Y) \rightarrow$
$\forall U$ s.t. $PSub(X,U)$ and $PSub(U,Y)$: $\neg PSub(U,Y)$ (*)

↳ $\forall X$ s.t. $CSub(A,X)$ and $CSub(X,B)$: $\neg CSub(X,B) \rightarrow$
$\forall Y,Z$ s.t. $Dom(Y,X)$ and $Dom(Z,B)$: $\neg PSub(Y,Z)$ (*) $\rightarrow$
$\forall U$ s.t. $PSub(Y,U)$ and $PSub(U,Z)$: $\neg PSub(U,Z)$ (**)

↳ $\forall X$ s.t. $CSub(A,X)$ and $CSub(X,B)$: $\neg CSub(X,B) \rightarrow$
$\forall Y,Z$ s.t. $Rng(Y,X)$ and $Rng(Z,B)$: $\neg PSub(Y,Z)$ (*) $\rightarrow$
$\forall U$ s.t. $PSub(Y,U)$ and $PSub(U,Z)$: $\neg PSub(U,Z)$ (**)

Proof ok for this case.

• **For** $SetUp(\mathbb{G},\mathbb{R},PSub(A,B))$**.** (Insertion of a sub-property relation.)

***PSub(A,B)* :**

↳ if $\exists DB, RB$ s.t. $\{Pr(B), Dom(B,DB), Rng(B,RB)\} \subseteq \mathbb{G}$ then do nothing else (do the update $Pr(B, Resource, Resource)$):
- ↳ $\neg Cl(B)$
  - ↳ $\forall X \neg Dom(X,B) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y,Z \neg PI(Y,Z,X)(*)$

↳ $\forall X \neg Rng(X,B) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y,Z \neg PI(Y,Z,X)(*)$
- ↳ $\neg Indiv(B)$
- ↳ $Pr(B)$
- ↳ $Dom(B, Resource)$, (note $DB = Resource$)
- ↳ $Rng(B, Resource)$, (note $RB = Resource$)

↳ if $\exists DA, RA$ s.t. $\{Pr(A), Dom(A,DA), Rng(A,RA)\}$ then
- ↳ if $DA = DB$ or $CSub(DA,DB)$ then do nothing
  else if $DA = Resource$ then
  - ↳ $\neg Pr(A) \rightarrow \forall X,Y \neg Pi(X,Y,A)$
  - ↳ $Pr(A)$
  - ↳ $Dom(A,DB)$
  - ↳ $Rng(A,RB)$

  else
  - ↳ $CSub(DA,DB)$ we insert here the tree corresponding to insert a sub class given above.
- ↳ if $RA = RB$ or $CSub(RA,RB)$ then do nothing
  else if $RA = Resource$ or $RA = Literal$ then
  - ↳ $\neg Pr(A) \rightarrow \forall X,Y \neg Pi(X,Y,A)$
  - ↳ $Pr(A)$
  - ↳ $Dom(A,DB)$
  - ↳ $Rng(A,RB)$

  else
  - ↳ $CSub(RA,RB)$ we insert here the tree corresponding to insert a sub class given above.

↳ if $\nexists DA, RA$ s.t. $\{Pr(A), Dom(A,DA), Rng(A,RA)\} \subseteq \mathbb{G}$ then
- ↳ $\neg Cl(A)$
  - ↳ $\forall X \neg Dom(X,A) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y,Z \neg PI(Y,Z,X)(*)$
  - ↳ $\forall X \neg Rng(X,A) \rightarrow \neg Pr(X)(*) \rightarrow \forall Y,Z \neg PI(Y,Z,X)(*)$
- ↳ $\neg Indiv(A)$
- ↳ $Pr(A)$
- ↳ $Dom(A,DB)$
- ↳ $Rng(A,RB)$

Proof ok for this case.

• **For** $SetUp(\mathbb{G}, \mathbb{R}, \neg PSub(A,B))$**.** (Deletion of a sub-property relation.)
$\neg PSub(A,B)$:

↳ $\forall X$ s.t. $PSub(A,X)$ and $PSub(X,B)$: $\neg PSub(X,B)$

Proof ok for this case.

D Side-effects (UpdCond table)

This is the full UpdCond table that is used to determine wich side effects need to be applied for insertion (table 2) and deletions (table 3) updates.

Update	Side effects on graph schema	Rule
Cl(A)	¬Pr(A)	4
	¬Indiv(A)	5
	CSub(A, Resource)	13
	Uri(A)	1
Pr(P)	¬Cl(P)	4
	¬Indiv(P)	6
	Dom(P, Resource)	15
	Rng(P, Resource)	15
	Uri(P)	3
Indiv(I)	¬Cl(I)	5
	¬Pr(I)	6
	CI(I, Resource)	14
	Uri(I)	3
CSub(XC,YC)	Cl(XC)	7
	Cl(YC) ∨ (YC= Ressource)	7
	∀ ZC such that CSub(YC,ZC) then CSub(XC,ZC)	18
	∀ ZC such that CSub(ZC,XC) then CSub(ZC,YC)	18
	¬CSub(YC,XC) % Error message contradiction, *i.e.* Csub(XC,XC)	19
	∀Zi such that CI(Zi, XC) then CI(Zi,YC)	26
PSub(Xp,Yp)	Pr(Xp)	8
	Pr(Yp)	8
	∀ Zp such that PSub(Yp,Zp) then PSub(Xp,Zp)	20
	∀ Zp such that PSub(Zp,Xp) then PSub(Zp,Yp)	20
	¬PSub(Yp,Xp) % Error message, *i.e.* Psub(Xp,Xp)	21
	∀ Zi1, Zi2 such that PI(Zi1, Zi2, Xp) then PI(Zi1, Zi2 ,Yp)	27
	Let Dom(Xp,ZD1) and Dom(Yp, ZD2) then ZD1=ZD2 or CSub(ZD1, ZD2)	22
	Let Rng(Xp,ZD1) and Rng(Yp, ZD2) then ZD1=ZD2 or CSub(ZD1, ZD2)	23
Dom(Xp, XD)	Pr(Xp)	9
	Cl(XD) ∨ (XD= Ressource)	9
	∀ YD ≠ XD , ¬ Dom (Xp,YD)	16
	∀ Xp1, XD1 such that PSub (Xp, Xp1) and Dom(Xp1, XD1) then CSub(XD, XD1) ∨ (XD = XD1)	22
	∀ Xp1, XD1 such that PSub (Xp1, Xp) and Dom(Xp1, XD1) then CSub(XD1, XD) ∨ (XD = XD1)	22
	∀ Xi, Yi such that PI(Xi, Yi, Xp) then CI(Xi, XD)	24
Rng(Xp, XR)	Pr(Xp)	10
	Cl(XR) ∨ (XR= Ressource) ∨ (XR is litteral)	10
	∀ YR ≠ XR , ¬ Dom (Xp,YR)	17
	∀ Xp1, XR1 such that PSub (Xp, Xp1) and Dom(Xp1, XR1) then CSub(XR, XR1) ∨ (XR = XR1)	23
	∀ Xp1, XR1 such that PSub (Xp1, Xp) and Dom(Xp1, XR1) then CSub(XR1, XR) ∨ (XR = XR1)	23
	∀ Xi, Yi such that PI(Xi, Yi, Xp) then CI(Yi, XR) ∨ (Lit(Yi) ∧ XR is litteral)	25
CI(Xi,XC)	Indiv (Xi)	11
	Cl (XC) ∨ (XC = Ressource)	11
	∀ YC CSub (XC, YC) then CI(Xi, YC)	26
PI(Xi,Yi, Xp)	Indiv (Xi)	12
	Indiv (Yi) ∨ Lit(Yi)	12
	Pr(Xp)	12
	∀ Yp PSub (Xp, Yp) then PI(Xi, Yi, Yp)	27
	Let Dom (Xp, XD) then CI(Xi, XD)	24
	Let Rng (Xp, XR) then CI(Yi, XR) ∨ (Lit(Yi) ∧ XR is litteral)	25

Table 2. UpdCond table for insertions.

Update	Side effects on graph schema	Rule
¬ Cl(A)	∀ Xsc CSub(Xsc, A) then ¬ CSub(Xsc, A)	7
	∀ XC CSub(A, XC) then ¬ CSub(A, XC)	7
	∀ XD Dom(XD, A) then ¬ Dom(XD, A)	9
	∀ XR Rng(XR, A) then ¬ Rng(XR, A)	10
	∀ Xi CI(Xi, A) then ¬ CI(Xi, A)	11
¬ Pr(P)	∀ Xsp PSub(Xsp, P) then ¬ PSub(Xsp, P)	8
	∀ XP PSub(P, XP) then ¬ PSub(P, XP)	8
	∀ XD Dom(P, XD) then ¬ Dom(P, XD)	9
	∀ XR Rng(P, XR) then ¬ Rng(P, XR)	10
	∀ Xi, Yi PI(Xi, Yi, P) then ¬ PI(Xi, Yi, P)	12
¬ Indiv(I)	∀ XC CI(I, XC) then ¬ CI(I, XC)	11
	∀ Xi, XP PI(I, Xi, XP) then ¬ PI(I, Xi, XP)	12
	∀ Xi, XP PI(Xi,I, XP) then ¬ PI(Xi,I, XP)	12
¬ CSub(Xsc,XC)	if XC = Resource then ¬ Cl(Xsc)	13
	NON deterministe SITUATIONS:	
	∀ Y such that CSub(Xsc, Y) and CSub(Y,XC) then	
	Choice (¬ CSub(Xsc,Y), ¬ CSub(Y, XC), both) or exception	18
	∀ XP1, XP2 such that PSub(XP1, XP2) and Dom(XP1,Xsc) and Dom (XP2,XC) then	
	Choice (¬ PSub(XP1, XP2), ¬ Dom(XP1,Xsc), ¬ Dom (XP2,XC), all them) or exception	22
	∀ XP1, XP2 such that PSub(XP1, XP2) and Rng(XP1,Xsc) and Rng (XP2,XC) then	
	Choice (¬ PSub(XP1, XP2), ¬ Rng(XP1,Xsc) ¬ Rng (XP2,XC), all them) or exception	23
¬ PSub(Xsp,XP)	NON deterministe SITUATION:	
	∀ Y such that PSub(Xsp, Y) and PSub(Y,XP) then	
	Choice (¬ PSub(Xsp,Y), ¬ PSub(Y, XP), both) or exception	20
¬ Dom(Xp, XD)	¬ Pr(Xp)	15
¬ Rng(Xp, XR)	¬ Pr(Xp)	15
¬ CI(Xi,XC)	if XC = ressource then ¬ Indiv(Xi)	14
	NON deterministe SITUATIONS:	
	∀ YC such that CI(Xi, YC) and CSub(YC,XC) then	
	Choice (¬ CI(Xi,YC), ¬ CSub(YC, XC), both) or exception	26
	∀ Yi, Zp such that PI(Xi, Yi, Zp) and Dom(Zp,XC) then	
	Choice (¬ PI(Xi, Yi, Zp), ¬ Dom(Zp,XC), both) or exception	24
	∀ Yi, Zp such that PI(Xi, Yi, Zp) and Rng(Zp,XC) then	
	Choice (¬ PI(Xi, Yi, Zp), ¬ Rng(Zp,XC), both) or exception	25
	¬ PI(Xi,Yi, Xp)	
	NON deterministe SITUATIONS:	
	∀ Yp such that PI(Xi,Yi, Yp) and PSub(Yp,Xp) then	
	Choice (¬ PI(Xi,Yi, Yp), ¬ PSub(Yp,Xp), both) or exception	27

Table 3. UpdCond table for deletions.