Specification of side-effect management techniques for semantic graph sanitization

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

PROJET SENDUP
ANR-18-CE23-0010

Specification of side-effect management techniques for semantic graph sanitization

DELIVERABLE D6

Related task	Task 2
Partner	LIFO
Redactor	Cédric Eichler
Contributors	Jacques Chabin, Cédric Eichler, Mirian Halfeld-Ferrari, Nicolas Hiot
Versioning	24/09/2020, V2: formatting 27/08/2020, V1: initial report

Presentation of this deliverable

The goal of the SENDUP project is to propose anonymisation mechanisms for data organized as graphs with an underlying semantic. Such mechanisms triggers updates on the database. This deliverable presents the update approach and side-effect management techniques defined in SENDUP.

We focus on updates -instance or schema changes- on RDF/S databases which are expected to satisfy RDF intrinsic semantic constraints. We model RDF/S databases as type graphs and use graph rewriting rules to formalize updates. Such rules define both the effect of a graph transformation and its applicability conditions. We propose 19 rules modelling atomic updates and prove that their application necessarily preserves the database's 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. These techniques are implemented in a dedicated software module S1 called SetUp. This deliverable also presents a preliminary experimental validation and evaluation of SetUp.

Contents

Chapter 1

Introduction

One of the objectives of SENDUP is to introduce techniques for the management of graph database updates to support their sanitization. Graph rewriting concerns the technique of transforming a graph. It is thus natural to conceive its application in the evolution of graph databases. This report leverage this formal tool within a framework that ensures the *consistent* evolution of RDF (Resource Description Framework) databases.

Initially just a part of the semantic web stack, RDF is currently largely used for representing high-quality connected data. "Representing data in RDF, in an integrated way, allows information to be identified, disambiguated and interconnected by software agents and various systems to read, analyze and act upon" [1]. Data should above all else be usable and therefore satisfy the various semantics and constraints requirement applications may have.

In the last decade, ontology-based systems have addressed knowledge representation by following the Open World Assumption (OWA) semantics where a statement cannot be inferred as false on the basis of failures to prove it. In SENDUP, we consider databases satisfying integrity constraints (IC) and the Closed World Assumption (CWA) semantics. Indeed, the OWA is not adapted to data-centric applications needing complete and valid knowledge [36]. On the contrary, the CWA allows a finer comprehension of the curated information and the released data, a crucial point in data sanitization. For example, a database where we want to ensure that *every drug is associated to a molecule* should be considered inconsistent if the drug d has not its associated molecule.

The following example illustrates our motivation and the challenges that pose the update of such databases.

Example 1 (Motivating Example.) *Fig. 2.2 shows a complete RDF/S graph database consistent w.r.t. to the RDF/S constraints. We are concerned by the problem of updating this database, keeping it consistent. Firstly, suppose an instance update: the insertion of ASA (acide amino-salicylique) as a class instance of Molecule. How can we guarantee that ASA will be also an instance of 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 report proposes SetUp (Schema Evolution Through UPdates), a maintenance tool based on graph rewriting rules for RDF data graph enriched with integrity constraints. Consistency is established according to the CWA semantics and ensures data quality for querying systems requiring reliable information and mastering released information. Constraints considered in this paper are those defining RDF/S semantics, but the approach adapts to other constraints, in particular user-defined ones. SetUp ensures sustainability since it offers the capability of dealing with evolution of data instance and structure without violating the semantics of the RDF model.

1.1 Characteristics of our solution

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 atomic type of 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. Graph rewriting rules ensure 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 rule. When an evolution is mandatory (for example when demanded by the anonymization process), 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 data graph which represents a set of RDF (the instance) and RDFS (the schema) triples respecting semantic constraints as defined in [8].
• 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. A dedicated section discuss future

methods of handling non-determinism.
• SetUp takes into account the user level. Only database administrators may require updates provoking schema changes.

1.2 Report Organization

The next chapter of this report sets up the work context. It introduce the notations, vocabulary and representations related to databases and updates used throughout the report.

Chapter 3 introduces the background on graph rewriting theory, focusing on the rewriting formalisms and approaches adopted in this report. This serve as a basis for chapter 4 where consistency-preserving rewriting rules formalizing atomic updates are specified. Such rules may not be applied if their application would lead to the introduction of inconsistencies.

In these cases, rather than refusing the update, we propose in Chapter 5 the generation of side-effects to force its application.

The rewriting rules formalizing graph updates and the techniques for the generation of side-effects have been implemented, resulting in a module called SetUpẆhile inherently part of the SENDUP software suite, SetUp can be used on its own through a dedicated API or TUI. Its experimental evaluation and validation is presented in Chapter 6.

Related work and the originality of our solution is discussed in Chapter 7. Chapter 8 offers concluding remarks, summarizing the proposed framework, its anticipated usage and its appropriateness with regard to SENDUP scenarios.

Chapter 2

RDF databases and updates

A collection of RDF statements intrinsically represents a typed attributed directed multi-graph, making the RDF model suited to certain kinds of knowledge representation [2]. Constraints on RDF facts can be expressed in RDFS (Resource Description Framework Schema), the schema language of RDF. RDF/S databases are formalized in two ways in this paper: as classical triple-based RDF statements and as a typed graph. This chapter introduces these representations, as well as the considered constraints.

2.1 Logical representation of RDF/S databases

In [8] we find a set of logical rules expressing the semantics of RDF/S (rules concerning RDF or RDFS) models. Let $\mathbf{A}_C$ and $\mathbf{A}_V$ be disjoint countably infinite sets of constants and variables, respectively. A *term* is a constant or a variable. Predicates are classified into two sets: (i) $\textsc{SchPred} = \{CL, Pr, CSub, Psub, Dom, Rng\}$, used to define the database schema, standing respectively for classes, properties, sub-classes, sub-properties, property domain and range, and (ii) $\textsc{InstPred} = \{CI, PI, Ind\}$, used to define the database instance, standing respectively for class and property instances and individuals. An *atom* has the form $P(u)$, where 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.

2.2 RDF/S databases as a typed graph

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 2.1 describes how each RDF triples are formalized and represented in the typed graph model.

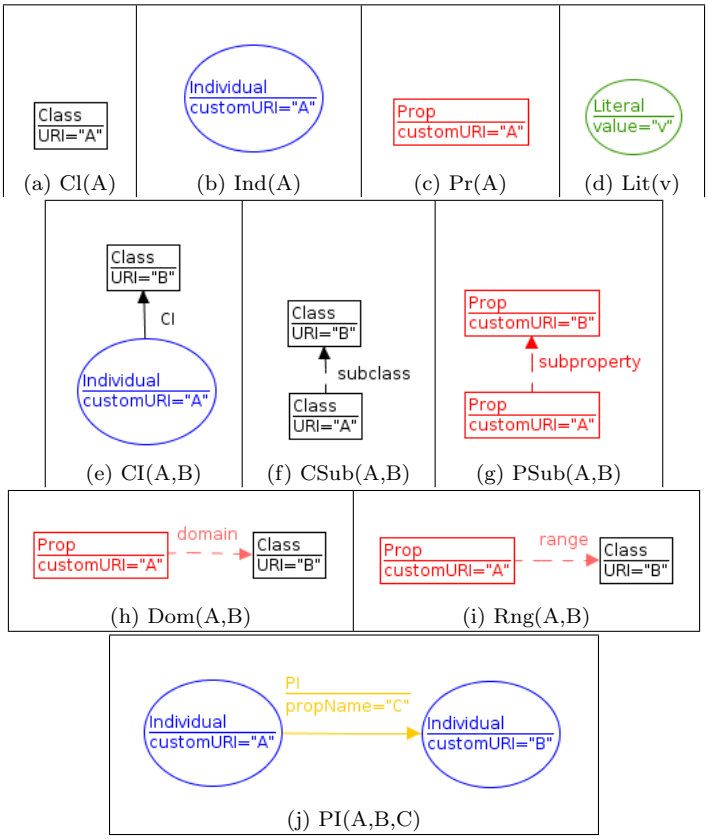


Figure 2.1: RDF triples in the type graph model

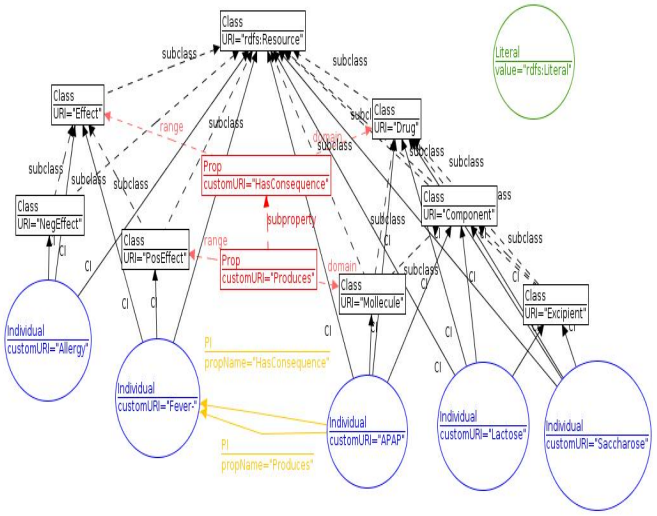


Figure 2.2: RDF schema and instance.

2.3 Example

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*). We note $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ the typed graph that represents the same database. 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.* □

Fig. 2.2 shows an RDF instance and schema as a typed graph. The schema specifies that *Has Consequence* is a property having class *Drug* as its domain and the class *Effect* as its range. Property *Produces* is a sub-property of *Has Consequence* while *PosEffect* is a sub-class of *Effect*. Class *"rdfs:Resource"* symbolizes the root of an RDF class hierarchy. The 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 $Fever^-$).

The logical representation of this database is a set of facts. For instance facts such as *CL(Drug)* or *CSub(Drug, rdfs:Resource)* are for the schema description and *Ind(Saccharose)* or *CI(Saccharose, Excipient)* are for the instance description.

• **Typing Constraints:**

$$CL(x) \Rightarrow URI(x) \quad (2.1) \qquad Pr(x) \Rightarrow URI(x) \quad (2.2)$$

$$Ind(x) \Rightarrow URI(x) \quad (2.3) \qquad (CL(x) \wedge Pr(x)) \Rightarrow \bot \quad (2.4)$$

$$(CL(x) \wedge Ind(x)) \Rightarrow \bot \quad (2.5) \qquad (Pr(x) \wedge Ind(x)) \Rightarrow \bot \quad (2.6)$$

$$CSub(x,y) \Rightarrow CL(x) \wedge CL(y) \quad (2.7) \qquad PSub(x,y) \Rightarrow Pr(x) \wedge Pr(y) \quad (2.8)$$

$$Dom(x,y) \Rightarrow Pr(x) \wedge CL(y) \quad (2.9) \qquad Rng(x,y) \Rightarrow Pr(x) \wedge CL(y) \quad (2.10)$$

$$CI(x,y) \Rightarrow Ind(x) \wedge CL(y) \quad (2.11) \qquad CL(x) \Rightarrow CSub(x, \textit{rdfs:Resource}) \quad (2.12)$$

$$Ind(x) \Rightarrow CI(x, \textit{rdfs:Resource}) \quad (2.13) \qquad PI(x,y,z) \Rightarrow Ind(x) \wedge Ind(y) \wedge Pr(z) \quad (2.14)$$

• **Schema Constraints:**

$$Pr(x) \Rightarrow (\exists y, z)(Dom(x,y) \wedge Rng(x,y)) \quad (2.15)$$

$$Pr(x) \Rightarrow \wedge Dom(x,y) \implies ((y \neq z) \wedge Dom(x,z)) \Rightarrow \bot \quad (2.16)$$

$$Pr(x) \Rightarrow \wedge Rng(x,y) \implies ((y \neq z) \wedge Rng(x,z)) \Rightarrow \bot \quad (2.17)$$

$$CSub(x,y) \wedge CSub(y,z) \Rightarrow CSub(x,z) \quad (2.18)$$

$$CSub(x,y) \wedge CSub(y,x) \Rightarrow \bot \quad (2.19)$$

$$PSub(x,y) \wedge PSub(y,z) \Rightarrow PSub(x,z) \quad (2.20)$$

$$Psub(x,y) \wedge Dom(x,z) \wedge Dom(y,w) \wedge (z \neq w) \Rightarrow CSub(z,w) \quad (2.21)$$

$$PSub(x,y) \wedge PSub(y,x) \Rightarrow \bot \quad (2.22)$$

$$Psub(x,y) \wedge Rng(x,z) \wedge Rng(y,w) \wedge (z \neq w) \Rightarrow CSub(z,w) \quad (2.23)$$

• **Instance Constraints:**

$$Dom(z,w) \Rightarrow (PI(x,y,z) \Rightarrow CI(x,w)) \quad (2.24)$$

$$Rng(z,w) \Rightarrow (PI(x,y,z) \Rightarrow CI(x,w)) \quad (2.25)$$

$$CSub(y,z) \Rightarrow (CI(x,y) \Rightarrow CI(x,z)) \quad (2.26)$$

$$PSub(z,w) \Rightarrow (PI(x,y,z) \Rightarrow PI(x,y,w)) \quad (2.27)$$

Figure 2.3: Simplified and compacted form of RDF/S constraints

2.4 Considered constraints

Constraints presented in [8] are those in Fig. 2.3 which is borrowed from [15]. We recall from [8] that these constraints capture the RDF/S semantics and the restrictions imposed by [30] whose model's goal is to provide sound and complete algorithm for RDF/S query containment and minimization. That model imposes a semantics having characteristics such as: role distinction between types (classes, properties and individuals), unique domains and ranges for properties and no cycles in subsumptions. These constraints (that we denote by $\mathscr{C}$) are the basis of our RDF semantics. For instance, the schema constraint (2.20) establishes transitivity between sub-properties and the instance constraint (2.27)

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 $\mathscr{C}$. Indeed, in accordance to the closed world assumption (CWA), constraints are not just inferences - they impose data restrictions.

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. 2.3).* □

2.5 Updates: definition and objectives

We define updates as follows:

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

Note that some updates may contain a contradiction, in which case they are intrinsically inconsistent.

Definition 4 (Intrinsically inconsistent update) *An intrinsically inconsistent update U related to a fact F is such that, $\forall \mathscr{D}, F \in \mathscr{D} \implies \neg(\mathscr{D}, \mathscr{C})$.*

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, qualifying a set of atomic update type. Each update type, can be formalized by a rewriting rule r describing 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. 2.4: on the logical level, $(\mathscr{D}, \mathscr{C})$ expresses consistent databases; on the data graph level, $(\mathbb{G}, \mathbb{R})$ expresses graph evolution with rules in $\mathbb{R}$ encompassing constraints from $\mathscr{C}$.

$$\begin{array}{ccc} \mathscr{D} & \textit{satisfy constraints} & \mathscr{C} \\ \updownarrow & & \restriction\downarrow \\ \mathbb{G} & \textit{evolution guided by} & \mathbb{R} \end{array}$$

Figure 2.4: Rewriting rules $\mathbb{R}$ and constraints $\mathscr{C}$.

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 $(\mathscr{D}', \mathscr{C})$ for $\mathscr{D}'/\mathbb{G}'$.

Chapter 3

Preliminaries: graph rewriting

We briefly introduce the theoretical background on the formal graph rewriting approach adopted in this report: the Single Push Out (SPO) [22] approach.

Graph rewriting is a well-studied field for the formal specification of graph transformations [26]. 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 SENDUP, we adopt the SPO formalism [22] 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) [12], *Positive Application Conditions* (PACs), and *General Application Conditions* (GACs) [27].

3.1 The SPO approach

3.1.1 Specifying rewriting rules

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 (*i.e.*, an edge-preserving morphism m from an induced subgraph of L to R). [1]

An example of an SPO rule is specified in Fig. 3.1. 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

[1] To avoid multiplying notation, we use notation L and R for every rule, even those in the logical formalism, sometimes with an index indicating the rule name.

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.

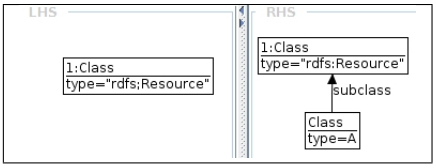


Figure 3.1: An SPO rewriting rule

3.1.2 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. 3.2. 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. 3.1 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.

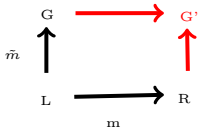


Figure 3.2: Push-Out, application of SPO rules

3.2 Extensions to restrict applicability

3.2.1 Negative and Positive Application Conditions

NACs and PACs are well-studied 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

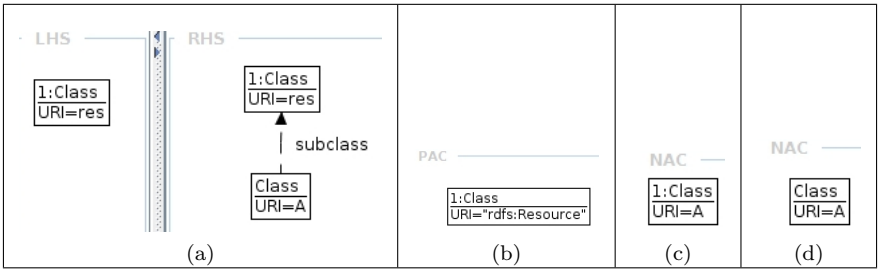


Figure 3.3: Less naive rewriting rule for the insertion of a class

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) $\bar{m} : P \to G$ (resp. $N \to G$) 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. 3.3 shows an enrichment of the rule depicted in Fig. 3.1. In the SPO core of the rule, the attribute "$rdfs : Resource$" is simply replaced by a variable res. The PAC specified in Fig. 3.3b 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. 3.1. In addition, it (i) avoids the addition of duplicate class node, thanks to the NAC of Fig. 3.3d 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. 3.3c, stating that the input A may not be equal to the res.

3.2.2 Nested Application Conditions, General Application Conditions

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* [10, 13] that allow 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

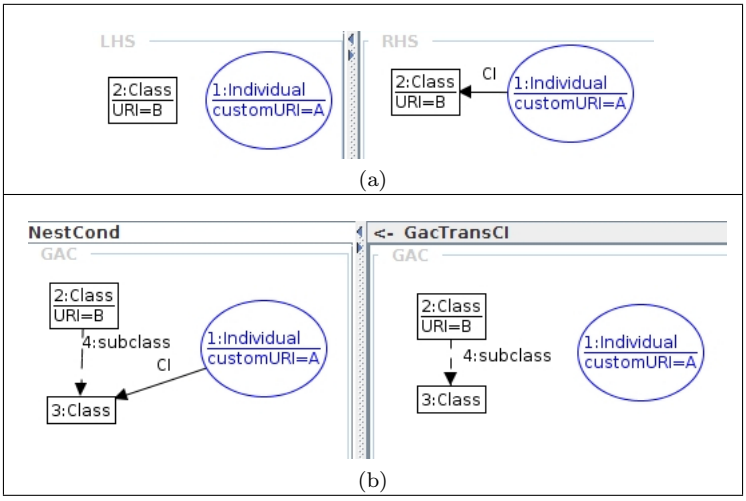


Figure 3.4: SPO rule for the insertion of a class instance with a GAC

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 [27] 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. 3.4 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. 3.4b. *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. 3.4b): *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 $Cl(C)$).

Chapter 4

Graph rewriting rules 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 Chapter 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 (2.11): 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 $CI(A, B)$ can be applied only on 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. This section presents rules of $\mathbb{R}$, together with proof of consistency preservation. Their presentation follows a standard basic form filled by the main explanations of the rule.

4.1 Insertion of a Class

Update category: Schema evolution
User level: Only authorized users such as database administrators or the anonymisation module
Rule semantics: This rule has been partially presented in Section 3 (Fig. 3.3).

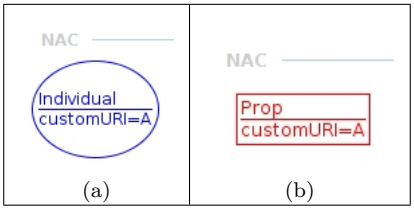


Figure 4.1: Additional NACs for addCl(A)

1) SPO specification: (Fig. 3.3a)
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. 3.3b) variable *res* is assigned to "*rdfs:Resource*"; this PAC corresponds to constraint (2.12).
3)NAC_{res} and NAC_{cl}: (Fig. 3.3d and 3.3c 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 4.1a and 4.1b, respectively): guarantee that the sets of classes, properties, and individuals are disjoint (constraints 2.4 and 2.5).

Proof of consistency preservation: It is clear from Fig. 2.3 that the addition of a class may activate constraints 2.4, 2.5, and 2.12 (*i.e.*, those having an atom with predicate *Cl* in their bodies). Thanks to the specification of NAC_{ind} and NAC_{pr}, constraints 2.4 and 2.5 are ensured. The PAC and SPO core of the rule in Fig. 3.3b and 3.3a impose the new class to be a subclass of *rdfs:Resource*, as constraint 2.12.

4.2 Deletion of a Class

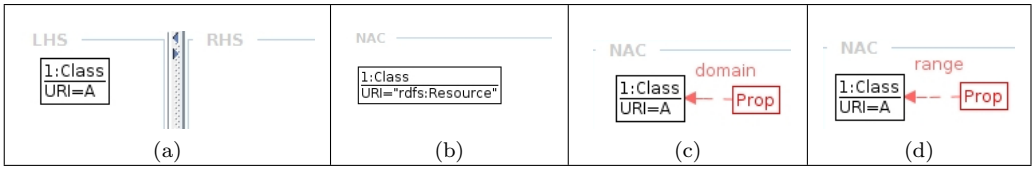


Figure 4.2: In (4.2a) the SPO rule for the deletion of a class, denoted $delCl(A)$. It has 3 NAC, namely (4.2b) NAC_{res} : $A \neq$ "$rdfs:Resource$"; (4.2c) NAC_{dom} : $\forall P$ such that $Pr(P)$, $Dom(P, A)$ is false; (4.2d) NAC_{rng}: $\forall P$ such that $Pr(P)$, $Rng(P, A)$ is false.

Update category: Schema evolution
User level: Only authorized users such as database administrators or the anonymi-

sation module
Rule semantics:
1) SPO specification: (Fig. 4.2a)
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. 4.2b) 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. 4.2c and 4.2d respectively) impose that the class being deleted is neither the domain nor the range of any property.

Proof of consistency preservation: From Fig. 2.3, the deletion of a class may impact constraints 2.7, 2.9, 2.10, 2.11 (those having $Cl(A)$ on the RHS) together with constraints 2.12, 2.13, 2.15 (as consequences of possible deletion politics). Constraints 2.7 and 2.11 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 2.9 (respect. 2.10) forbids the deletion of A as the domain (respect. as a range) of an existing property (which would also impact rule 2.15). 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 2.12 and 2.13 are never violated by the deletion of a class.

4.3 Insertion of a Class Instance

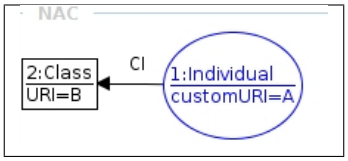


Figure 4.3: Additional NAC for addCI(A,B)

Update category: Instance evolution
User level: Any user
Rule semantics: This rule has been partially presented in Sec. 3 (Fig. 3.4).
1) SPO specification: (Fig. 3.4a)
LHS: There exists 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. 4.3) forbids the application of the rule if $CI(A, B)$ already exists in the database.
3)GAC: (Fig. 3.4b) 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. 2.3, constraints 2.11 and 2.26

(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 2.11). According to *GacTrans*, if there existss 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 2.26).

4.4 Deletion of a class instance

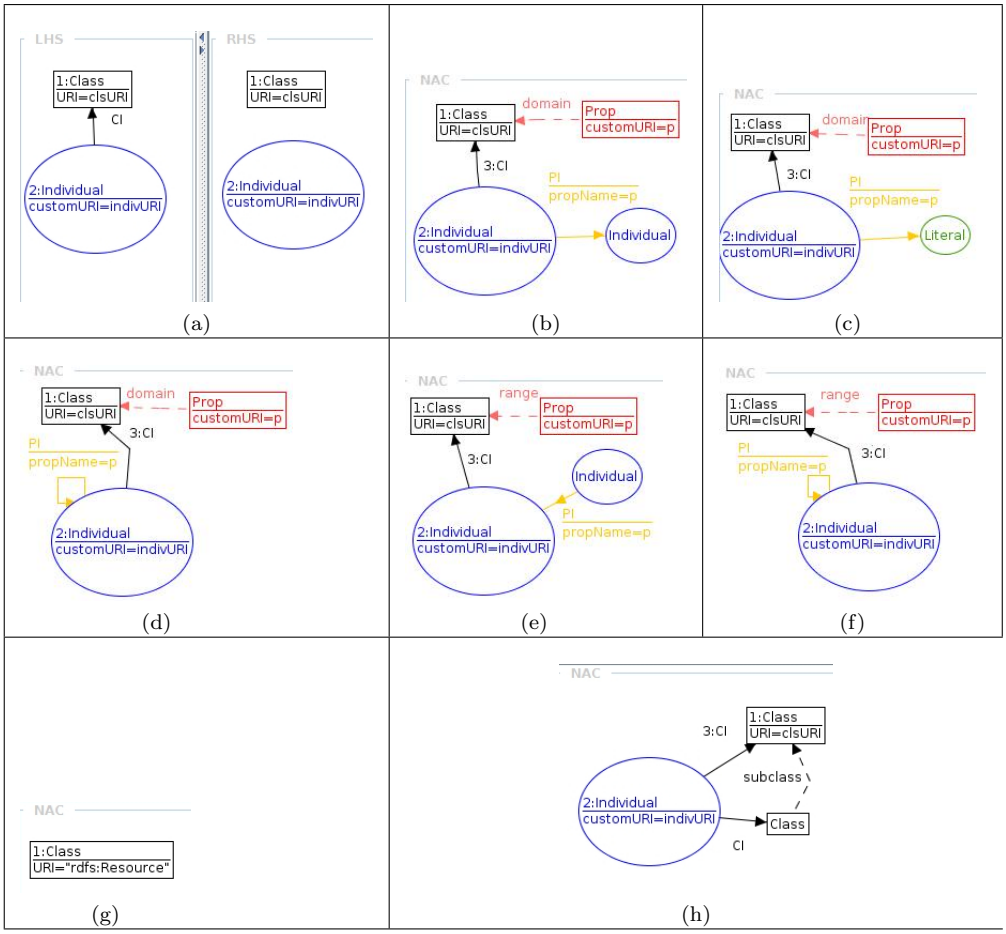


Figure 4.4: Rule concerning the deletion of a class instance.

Update category: Instance evolution
User level: Any user
Rule semantics:

1) SPO specification: (Fig. 4.4a)
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. 4.4b 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. 4.4c and 4.4d 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. 4.4f and 4.4g impose similar prohibition when the given class is the range of the property. The NAC in Fig 4.4g ensures that no instance of resource is removed – since an individual is always an instance of class Resource. The NAC in Fig 4.4h disallows the rule application when the individual is an instance of a subclass of the given class.

Proof of consistency preservation: From Fig. 2.3, we remark that constraints 2.13, 2.24, 2.25, and 2.26 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 4.4g ensures the satisfaction of constraint 2.13. The NACs in Figs. 4.4b–4.4f ensures the satisfaction of constraints 2.24 and 2.25. Finally the NAC in Fig 4.4h guarantees that constraint 2.26 is not violated.

4.5 Insertion of an individual

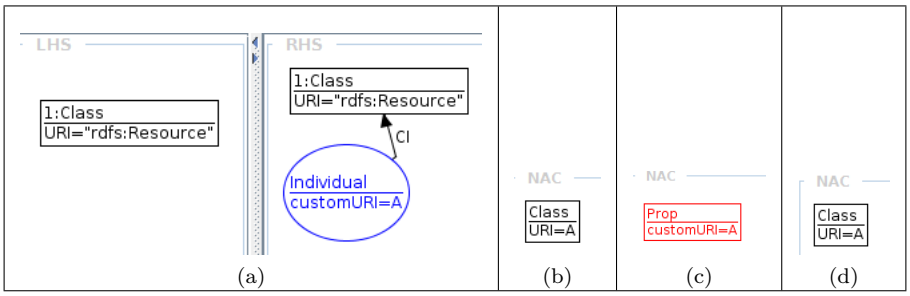


Figure 4.5: Rewriting rule for the insertion of an individual

Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 4.5a)
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. 4.5c and 4.5d guarantee that the sets of classes, properties, and individuals are disjoint (constraints 2.5 and 2.6 in Fig. 2.3). The Nac from Fig. 4.5b 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 2.3 (Fig. 2.3) requiring an URL (given as a rule parameter) and constraints 2.5 and 2.6 which are guaranteed by the two NACs. 4.5c and 4.5d. Unicity is guaranteed by NAC 4.5b .

4.6 Deletion of an individual

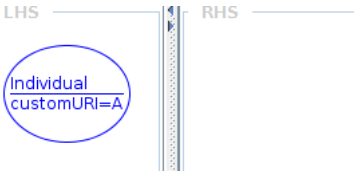


Figure 4.6: Rewriting rule for the deletion of an individual

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. 2.3, the deletion of an individual may impact constraints 2.11 and 2.14. These constraints are still preserved because CI 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.

4.7 Insertion of a literal

Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 4.7a)
LHS: Empty
RHS: The application of the rule inserts a node corresponding to the literal and

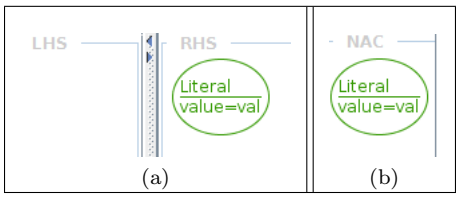


Figure 4.7: Rewriting rule for the insertion of a literal

its associated value in the graph.
2)NACs: (Fig. 4.7b) 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. 2.3), just allowing its future use –as a value for property for instance–. The NAC avoids literal redundancy.

4.8 Deletion of a literal

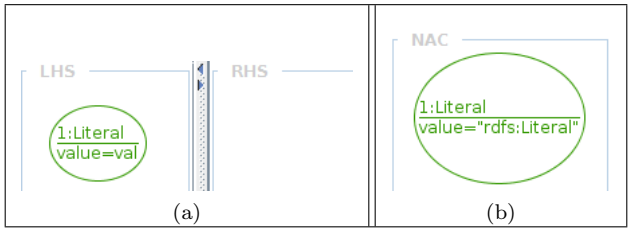


Figure 4.8: Rewriting rule for the deletion of a literal

Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 4.8a)
LHS: The node corresponding to the literal.
RHS: Empty.
2)NACs: (Fig. 4.8b) 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. 2.3, the deletion of a literal is only concerned by constraint 2.14 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.

4.9 Insertion of a property

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

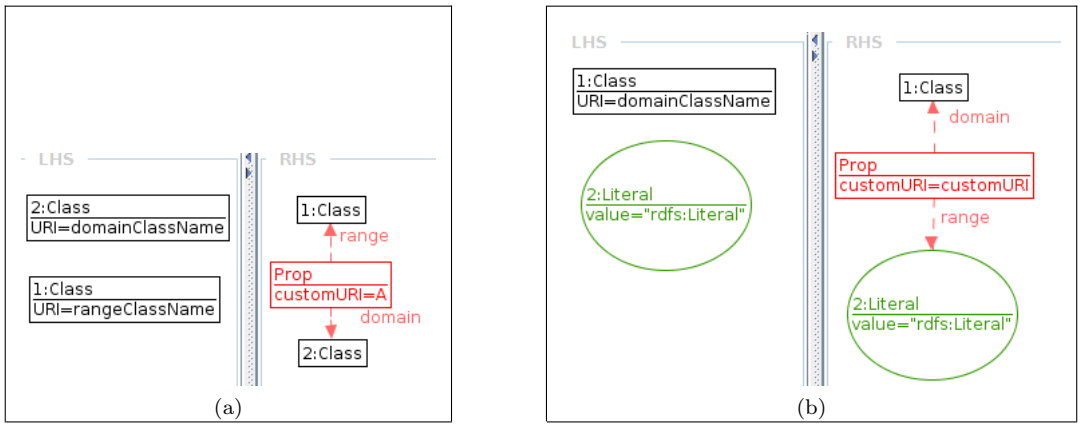


Figure 4.9: Rewriting rules for inserting properties come in two versions according to the the type of the property's range.

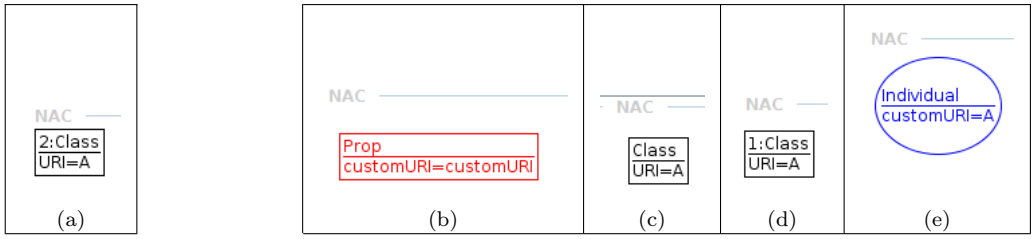


Figure 4.10: NACs for the insertion of a property.

4.9.1 Insertion of a property having a class as its range

Update category: Schema evolution
User level: Only authorized users such as database administrators or the anonymisation module
Rule semantics:
1) SPO specification: (Fig. 4.9a)
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. 4.10a, 4.10c and 4.10d guarantee that there exist no class with URI A (Fig. 4.10c), including the range (Fig. 4.10a) and the domain (Fig.4.10d) classes. NAC of Fig 4.10e prohibits the existence of an individual whose URI is A. Again, the NACs ensure that classes, properties, and individuals are disjoint sets (constraints 2.4 and 2.5 in Fig. 2.3). Finally, NAC 4.10b 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 2.2, 2.4 and 2.6 of Fig. 2.3. The NACs in Figs. 4.10a, 4.10c and 4.10d 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 2.15 in Fig. 2.3 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 2.15 is respected even when the same class is defined as the domain and the range of a given property.

4.9.2 Insertion of a property having a literal as its range

Update category: Schema evolution
User level: Only authorized users such as database administrators or the anonymisation module
Rule semantics:
1) SPO specification: (Fig. 4.9b)
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. 4.10d, 4.10c, 4.10e, and 4.10b. These two first NACs guarantee that there exist no class with URI A (Fig. 4.10c), including the domain (Fig.4.10d) class. NAC of Fig 4.10e prohibits the existence of an individual whose URI is A. Again, the NACs ensure that classes, properties, and individuals are disjoint sets (constraints 2.4 and 2.5 in Fig. 2.3). Finally, NAC 4.10b 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.

4.10 Deletion of a property

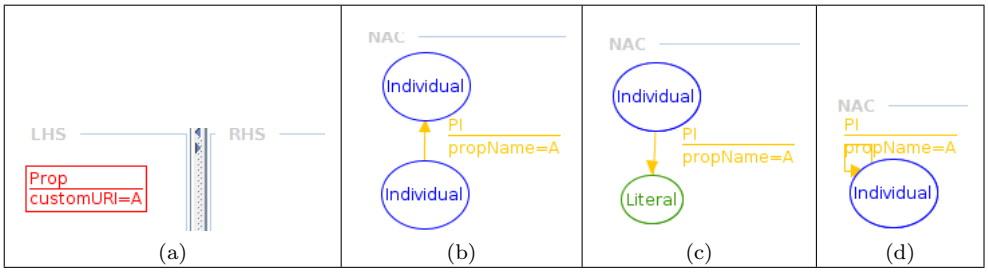


Figure 4.11: Rule concerning the deletion of a property (with associated NACs).

Update category: Schema evolution
User level: Only authorized users such as database administrators or the anonymisation module
Rule semantics:
1) SPO specification: (Fig. 4.11a)
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. 4.11b where the instances of the property are individuals), between an individual and a literal (Fig. 4.11c) or an atomic loop (Fig. 4.11d).

Proof of consistency preservation: From Fig. 2.3, we can remark that constraints 2.8, 2.9, 2.10 and 2.14 are concerned by the deletion of a property. Constraints 2.8, 2.9, 2.10 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 2.8), property domain (constraint 2.9) or property range (constraint 2.10). Constraint 2.14 is preserved because NACs prohibit the deletion of a property having instances.

4.11 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.

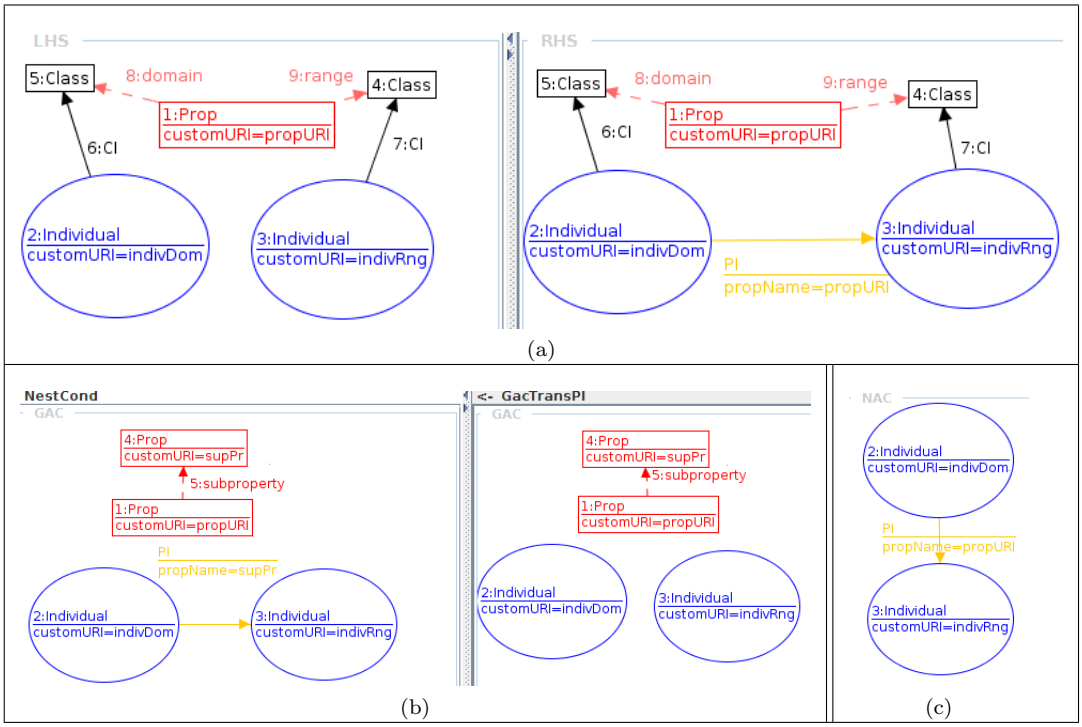


Figure 4.12: Rewriting rule for the insertion of a property instance when the property range is a class.

4.11.1 Insertion of a property instance for a property having a class as its range

Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 4.12a)
LHS: the LHS is composed of a property (identified by *propURI*) having two classes as its domain and range. Two individuals are declared as instances of each of these classes.
RHS: LHS plus an "property instance" edge attributed with *propURI* connecting the individuals. The edge represents manifests the presence of an instance of the property having the individuals as subject and object.
2)GAC: (Fig. 4.12b) 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. 4.12c) 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 2.14, 2.24 and 2.25 of Fig. 2.3 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 2.14), (ii) that the source of the PI is an instance of its class domain (constraint 2.24) and (iii) that the target of the PI is an instance of its class range (constraint 2.25). Constraint 2.27 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.

4.11.2 Insertion of a property instance for a property having a literal as its range

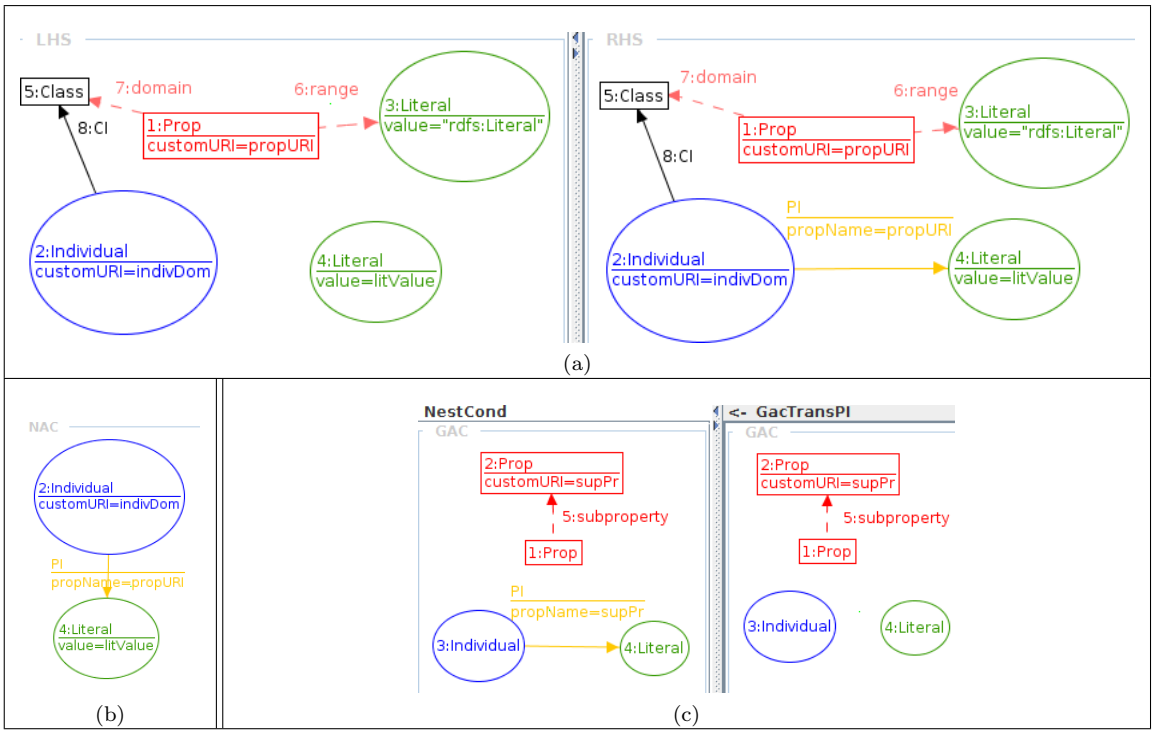


Figure 4.13: Rewriting rule for the insertion of a property instance when the property range is a literal.

Update category: Instance evolution

User level: Any user
Rule semantics:
1) SPO specification: (Fig. 4.13a)
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. 4.13c) 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. 4.13b) 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.

4.12 Deletion of a property instance

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

4.12.1 Deletion of a property instance for a property having a class as its range

Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 4.14a)
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. 4.14b) 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 2.27 of Fig. 2.3. The NAC ensures this constraint since the rule cannot be triggered if there exist sub-property instance links between the individuals.

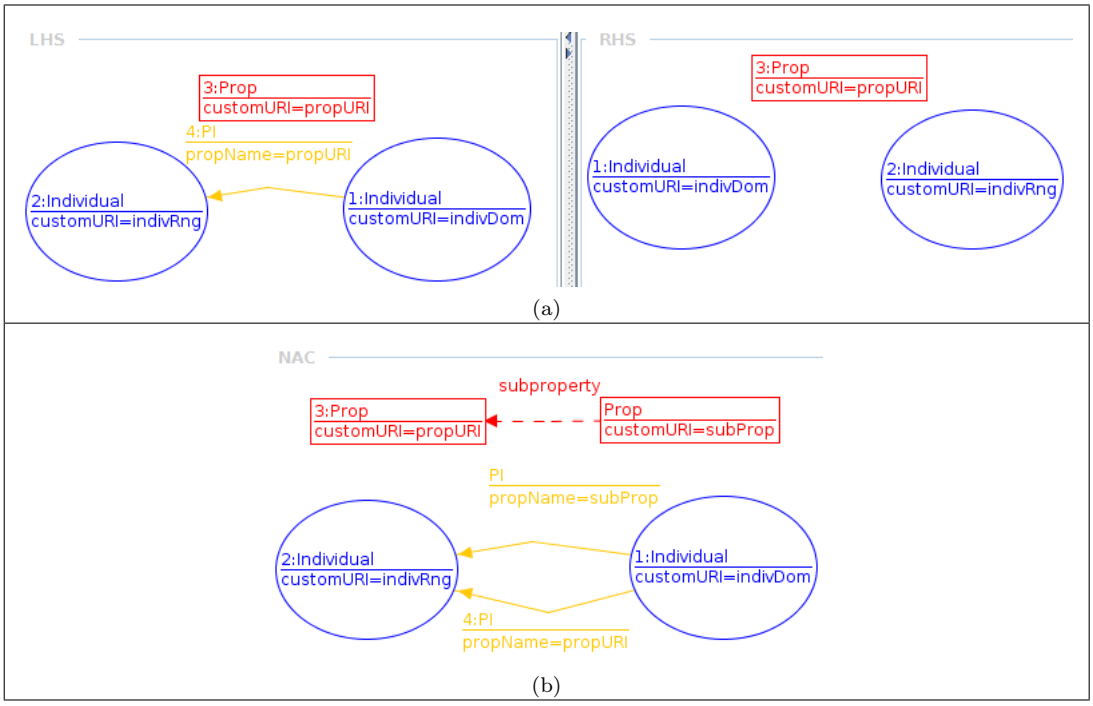


Figure 4.14: Rewriting rule for the deletion of a property instance when the property range is a class.

4.12.2 Deletion of a property instance for a property having a literal as its range

Update category: Instance evolution
User level: Any user
Rule semantics:
1) SPO specification: (Fig. 4.15a)
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 individual to the literal.
2)NACs: (Fig. 4.15b) 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.

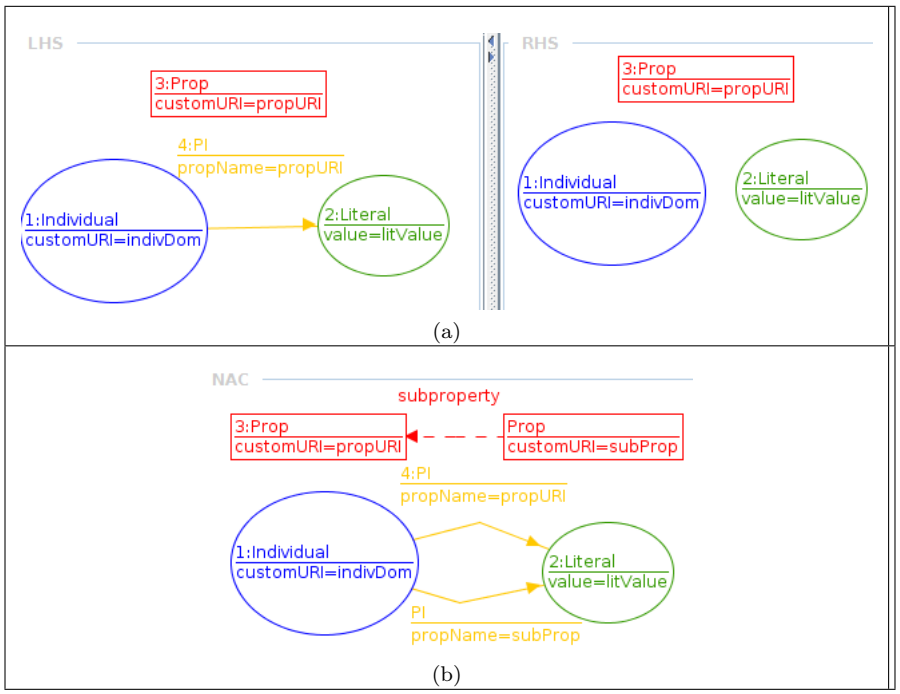


Figure 4.15: Rewriting rule for the deletion of a property instance when the property range is a literal.

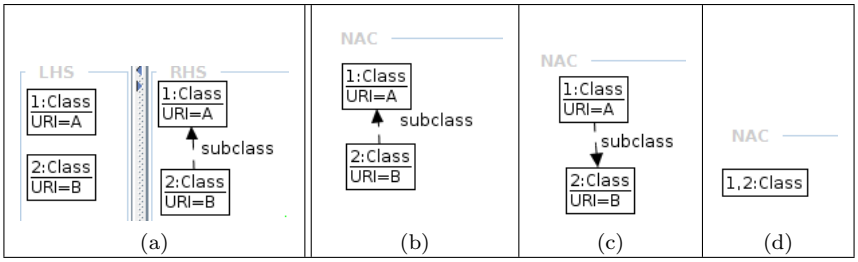


Figure 4.16: Rule concerning the insertion of a subclass relation (with associated NACs).

4.13 Insertion of a subclass relation

Update category: Schema evolution
User level: Only authorized users such as database administrators or the anonymisation module
Rule semantics:
1) SPO specification: Fig 4.16a

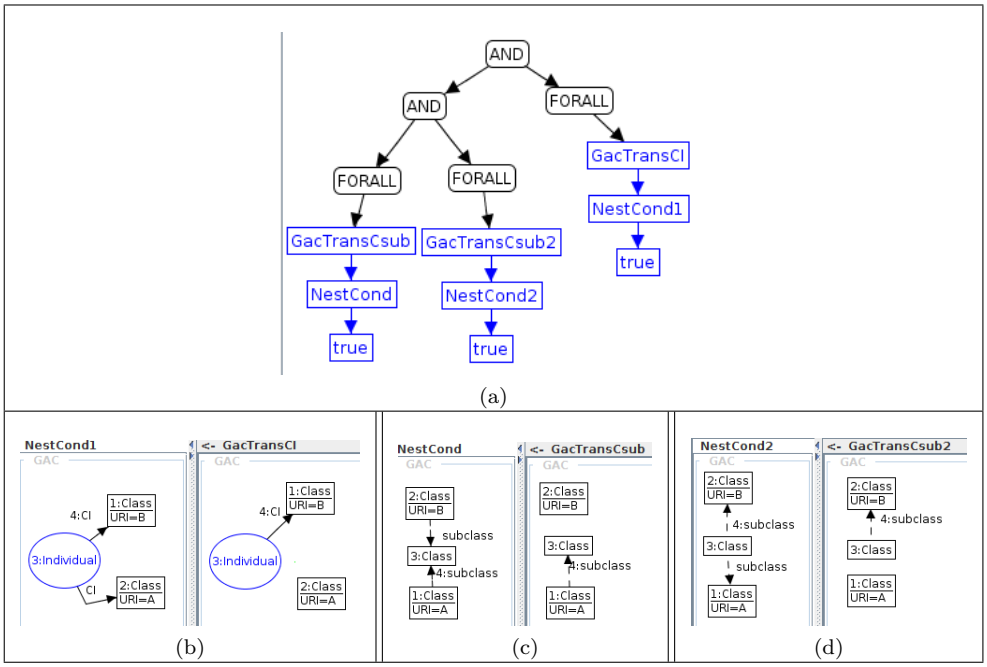


Figure 4.17: GAGs concerning the insertion of a sub-class property.

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 4.16b ensures that class B is not a subclass of class A yet, while the NAC in Fig 4.16c 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 4.16d forbids reflexive subclass relationship.
3) GACs: Fig. 4.17
The tree in Fig 4.17a 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. 4.16b. 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. 4.17 gathers two conditions. The leftmost one corresponds to Fig 4.17c. 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.4.17d. It states that all subclass of class B (`GacTransCsub2`) is a subclass of A (`NestCond2`).

Proof of consistency preservation: From Fig. 2.3, we remark that constraints 2.7, 2.18, 2.19 and 2.26 are concerned by the insertion of a subclass. Constraint 2.7 is not violated since the LHS of the SPO specification (Fig. 4.16a) imposes the existence of two classes before the addition of the edge representing the subclass relationship. GACs in Figs. 4.17c and 4.17d ensures the satisfaction of constraint 2.18 of Fig. 2.3, since they guarantee the application of the rule only if the class hierarchy stays consistent. Constraint 2.19 is implemented by the NACs which ensure that a cyclic subclass hierarchy is not possible. Constraint 2.26 is ensured by GAC in Fig. 4.17b which imposes the application of the rule only if all instances of class B are already instances of class A.

4.14 Deletion of a subclass relation

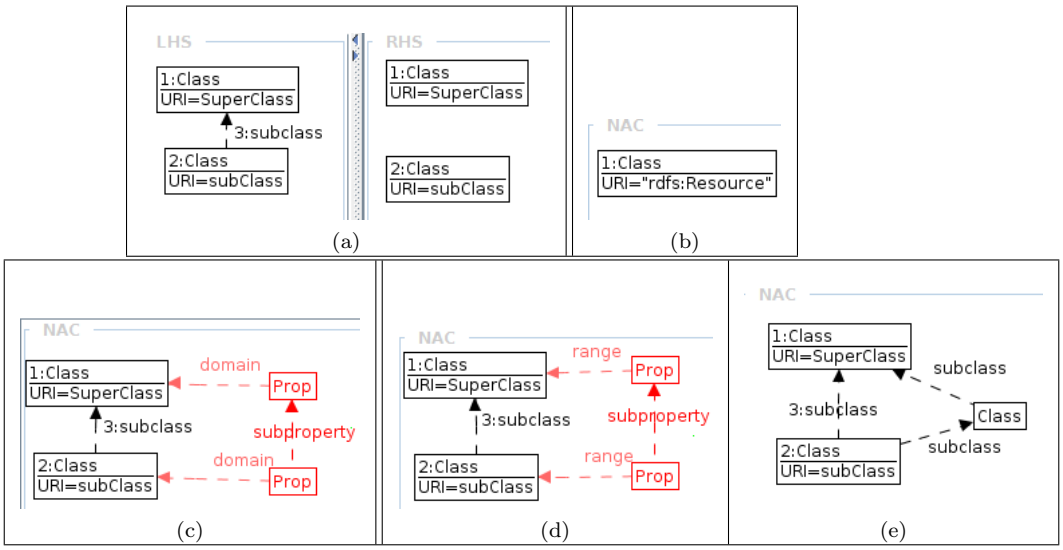


Figure 4.18: Rule concerning the deletion of a subclass relation (with associated NACs).

Update category: Schema evolution
User level: Only authorized users such as database administrators or the anonymisation module
Rule semantics:
1) SPO specification: Fig 4.18a
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 4.18b ensures that class *SuperClass* is not "rdfs:Resource", since all classes are subclasses of the root. The NAC in Fig 4.18c (resp. Fig 4.18d)

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 4.18e forbids the existence of a class that is both a subclass of *SuperClass* and a superclass of *subClass*, ensuring consistency with regard to transitivity.

Proof of consistency preservation: From Fig. 2.3, we remark that only constraints 2.12, 2.18, 2.21 and 2.23 are concerned by the deletion of a subclass relation. Constraint 2.12 is preserved thanks to the NAC defined in Fig. 4.18b that forbids the suppression of the sub-class relation to the root of the class hierarchy. The NAC of Fig. 4.18e ensures that the transitivity of the sub-class relation is respected, guaranteeing the respect of constraint 2.18. Finally, constraints 2.21 and 2.23 are ensured by NACs depicted in Figs. 4.18c and 4.18d, 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.

4.15 Insertion of a sub-property relation

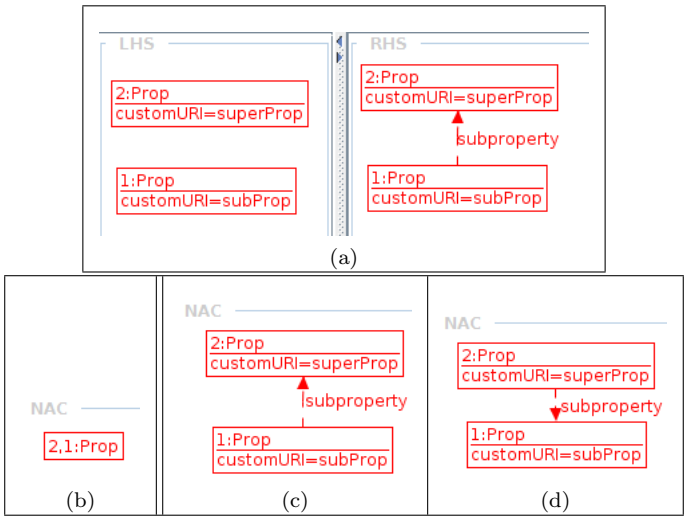


Figure 4.19: Rule concerning the insertion of a subproperty relation subclass (with associated NACs).

Update category: Schema evolution
User level: Only authorized users such as database administrators or the anonymisation module
Rule semantics:
1) SPO specification: Fig 4.19a

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 4.19b and 4.19d ensure that the sub-property relation is neither reflexive nor symmetric, respectively. The NAC formalized in Fig. 4.19c forbids the insertion of the relation if it already exists.
3) GACs: The logical formula for the GACs is presented in Fig 4.20 while the GACs are formalized in Fig. 4.21 and 4.22. The logical formula states that all of the following conditions must be fulfilled for the rule to be applicable:

- $SameDom \vee SubDom$ (Fig. 4.21a and 4.21b); 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. 4.21c, 4.21d, and 4.21e); 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. 4.22a); all couple of individual linked with an instance of $superProp$ also have an instance of $subProp$.
- for all patterns $GacTransPISelf$, $NCselfPI$ is true (Fig. 4.22c); all individual with a reflexive instance of $superProp$ also has an instance of $subProp$.
- for all patterns $GacTransPILit$, $NCtransPIlit$ is true (Fig. 4.22b); 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.4.22d) (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. 2.3, we remark that only constraints 2.8, 2.20, 2.22, 2.27, 2.21, and 2.23 are concerned by the deletion of a subproperty relation.

The typing of the relation (constraint 2.8) are guaranteed by the SPO part of the rule that may match only property-typed nodes.

Constraints 2.22 is preserved thanks to the NACs defined in Fig. 4.19b and 4.19d that forbid the introduction of a cycle in the sub-property relation. The GACs of Fig. 4.22d and 4.22e ensure the preservation of the relation transitivity (constraint 2.20).

The preservation of property instance propagation (constraint 2.27) is ensured by the GACs represented in Fig. 4.22c, 4.22a, and 4.22a.

Finally, constraints 2.21 and 2.23 are ensured by GACs depicted in Figs. 4.21a and 4.21b and 4.21c, 4.21d, and 4.21e, respectively. The sub-property relationship may be added only if the two properties have the same domain (resp. same

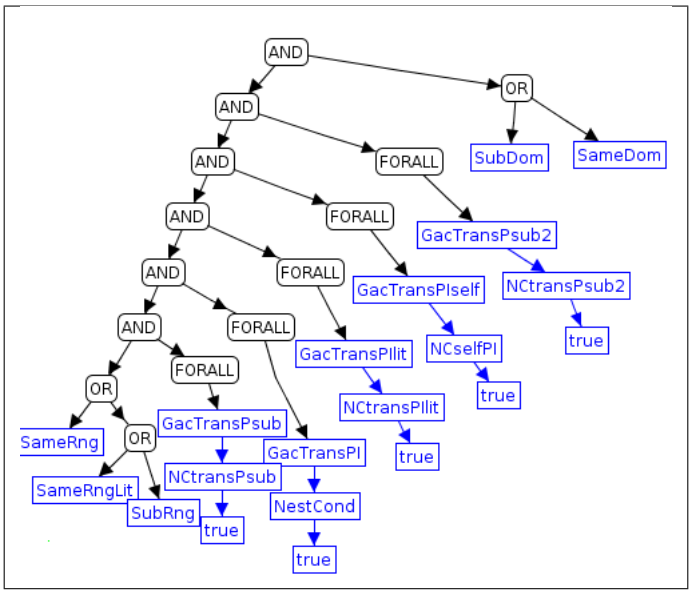


Figure 4.20: Logical relations for GACs regarding the insertion of a subproperty relation subclass.

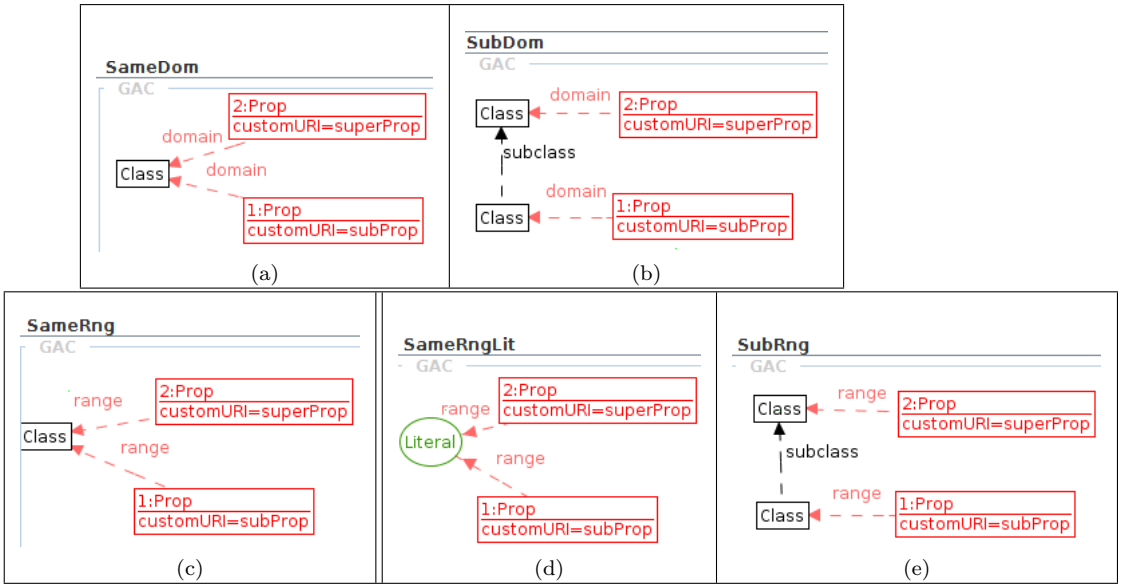


Figure 4.21: GACs for the insertion of a subproperty relation subclass.

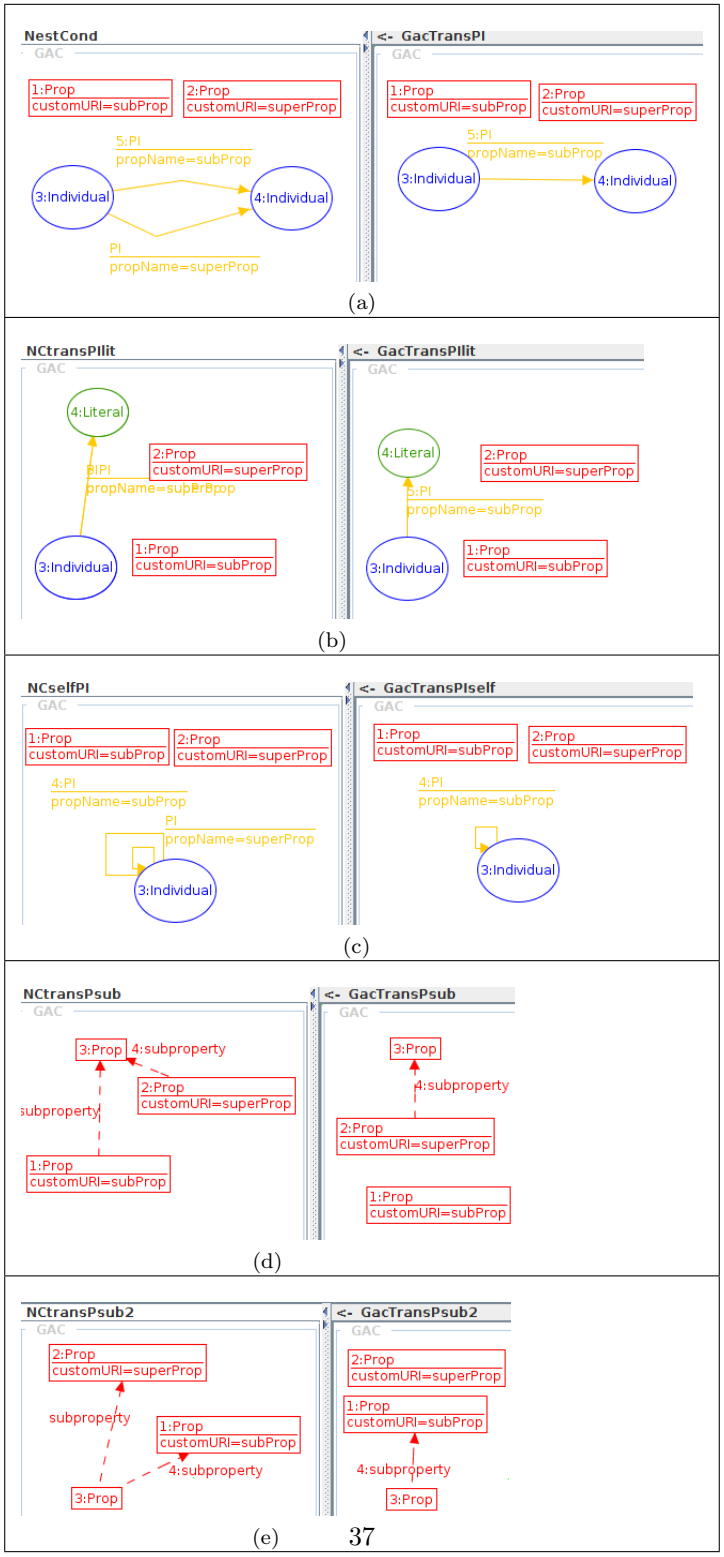


Figure 4.22: GACs for the insertion of a subproperty relation subclass (cont').

range) or if their respective domains (resp. ranges) are related with an adequate sub-class relationship.

4.16 Deletion of a subproperty relation

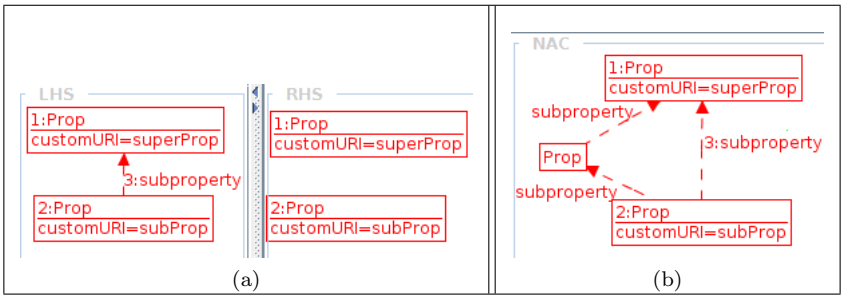


Figure 4.23: Rule concerning the deletion of a subproperty relation (with associated NAC).

Update category: Schema evolution
User level: Only authorized users such as database administrators
Rule semantics:
1) SPO specification: Fig 4.23a
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 4.23b 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. 2.3, we remark that only constraint 2.20 is concerned by the deletion of a suproperty relation. Its conservation is ensured by the NAC of Fig. 4.23b that forbids deletion of the relation if it has to exist due to transitivity.

4.17 Concluding remarks on consistent updates

In this chapter, we formalized 19 graph rewriting rules modelling atomic updates on RDF/S instance and schema. We demonstrated that each rule preserve RDF/S intrinsic constraints, *i.e.*if a rule is applicable to a consistent database, its application necessarily produce a consistent database. From each individual proof, we can derive the following lemma:

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 1 *Individual proofs have been provided above 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.*

In conclusion, rules of $\mathbb{R}$ allows consistent updates of RDF/S databases. However, when the application of a rule would introduce an inconsistency (*e.g.* suppressing a property which has instances), the rule cannot be applied. When updates are triggered by the anonymisation module, refusing an update is unacceptable. The following chapter presents our approach for the generation of compensation action guaranteeing that any rule may be forcefully applied (while still maintaining consistency).

Chapter 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, each update U is formalized by 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. The following example illustrates this approach.

Example 2 *Let $\mathscr{D}/\mathbb{G}$ be the database as the one in Fig. 2.2, 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 (Sec 4.3). 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. 2.2, except for a missing sub-class edge between Effect and NegEffect and a missing class instance edge from Allergy to Effect. □

This chapter introduces our technique for side-effect management including the side-effects triggered by each update and the corresponding algorithm, SetUp.

5.1 SetUp

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* such as the anonymization module.

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*$(\mathbb{G}, \mathbb{R}, U)$
2: **for all** condition c in *PreConditons* **do**
3: **if** c is not satisfied in $\mathbb{G}$ **then**
4: U' :=*Planner2FitGraphInCond* $(\mathbb{G}, c)$
5: **for all** update u' in U' **do**
6: $\mathbb{G}$:= SetUp $(\mathbb{G}, \mathbb{R}, u')$
7: $\mathbb{G}$:= *GraphRewriter*$(\mathbb{G}, \mathbb{R}, U)$
8: **return** $\mathbb{G}$

Given a database $\mathscr{D}/\mathbb{G}$ and an update U, Algorithm 1 transforms $\mathbb{G}$ by applying 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 is applicable.

5.1.1 Generating pre-condition

On line 1 of Algorithm 1, each condition c, necessary for applying r_U on $\mathbb{G}$, is added to *PreConditions*. Function *FindPredCond2ApplyUpd* works on table UpdCond indexed by the update type. Tables 5.2 and 5.1 show UpdCond for deletion and insertion, respectively. For example, from the penultimate row of Table 5.1, we know that the insertion of $CI(A, B)$, depends on the existence of A as an individual, B as a class and the respect of hierarchical constraints (B is a sub-class of the root hierarchy, and A is an instance of all super-classes of B).

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

Update	Side effects on graph schema	Rule
Cl(A)	¬Pr(A)	2.4
	¬Indiv(A)	2.5
	CSub(A, Resource)	2.12
	Uri(A)	2.1
Pr(P)	¬Cl(P)	2.4
	¬Indiv(P)	2.6
	Dom(P, Resource)	2.15
	Rng(P, Resource)	2.15
	Uri(P)	2.3
Indiv(I)	¬Cl(I)	2.5
	¬Pr(I)	2.6
	CI(I, Resource)	2.13
	Uri(I)	2.3
CSub(XC,YC)	Cl(XC)	2.7
	Cl(YC) ∨ (YC= Ressource)	2.7
	∀ ZC such that CSub(YC,ZC) then CSub(XC,ZC)	2.18
	∀ ZC such that CSub(ZC,XC) then CSub(ZC,YC)	2.18
	¬CSub(YC,XC) % Error message contradiction, *i.e.* Csub(XC,XC)	2.19
	∀Zi such that CI(Zi, XC) then CI(Zi,YC)	2.26
PSub(Xp,Yp)	Pr(Xp)	2.8
	Pr(Yp)	2.8
	∀ Zp such that PSub(Yp,Zp) then PSub(Xp,Zp)	2.20
	∀ Zp such that PSub(Zp,Xp) then PSub(Zp,Yp)	2.20
	¬PSub(Yp,Xp) % Error message, *i.e.* Psub(Xp,Xp)	2.22
	∀ Zi1, Zi2 such that PI(Zi1, Zi2, Xp) then PI(Zi1, Zi2 ,Yp)	2.27
	Let Dom(Xp,ZD1) and Dom(Yp, ZD2) then ZD1=ZD2 or CSub(ZD1, ZD2)	2.21
	Let Rng(Xp,ZD1) and Rng(Yp, ZD2) then ZD1=ZD2 or CSub(ZD1, ZD2)	2.23
Dom(Xp, XD)	Pr(Xp)	2.9
	Cl(XD) ∨ (XD= Ressource)	2.9
	∀ YD ≠ XD , ¬ Dom (Xp,YD)	2.16
	∀ Xp1, XD1 such that PSub (Xp, Xp1) and Dom(Xp1, XD1) then CSub(XD, XD1) ∨ (XD = XD1)	2.21
	∀ Xp1, XD1 such that PSub (Xp1, Xp) and Dom(Xp1, XD1) then CSub(XD1, XD) ∨ (XD = XD1)	2.21
	∀ Xi, Yi such that PI(Xi, Yi, Xp) then CI(Xi, XD)	2.24
Rng(Xp, XR)	Pr(Xp)	2.10
	Cl(XR) ∨ (XR= Ressource) ∨ (XR is litteral)	2.10
	∀ YR ≠ XR , ¬ Dom (Xp,YR)	2.17
	∀ Xp1, XR1 such that PSub (Xp, Xp1) and Dom(Xp1, XR1) then CSub(XR, XR1) ∨ (XR = XR1)	2.23
	∀ Xp1, XR1 such that PSub (Xp1, Xp) and Dom(Xp1, XR1) then CSub(XR1, XR) ∨ (XR = XR1)	2.23
	∀ Xi, Yi such that PI(Xi, Yi, Xp) then CI(Yi, XR) ∨ (Lit(Yi) ∧ XR is literal)	2.25
CI(Xi,XC)	Indiv (Xi)	2.11
	Cl (XC) ∨ (XC = Ressource)	2.11
	∀ YC CSub (XC, YC) then CI(Xi, YC)	2.26
PI(Xi,Yi, Xp)	Indiv (Xi)	2.14
	Indiv (Yi) ∨ Lit(Yi)	2.14
	Pr(Xp)	2.14
	∀ Yp PSub (Xp, Yp) then PI(Xi, Yi, Yp)	2.27
	Let Dom (Xp, XD) then CI(Xi, XD)	2.24
	Let Rng (Xp, XR) then CI(Yi, XR) ∨ (Lit(Yi) ∧ XR is literal)	2.25

Table 5.1: UpdCond table for insertions.

Update	Side effects on graph schema	Rule
¬ Cl(A)	∀ Xsc CSub(Xsc, A) then ¬ CSub(Xsc, A)	2.7
	∀ XC CSub(A, XC) then ¬ CSub(A, XC)	2.7
	∀ XD Dom(XD, A) then ¬ Dom(XD, A)	2.9
	∀ XR Rng(XR, A) then ¬ Rng(XR, A)	2.10
	∀ Xi CI(Xi, A) then ¬ CI(Xi, A)	2.11
¬ Pr(P)	∀ Xsp PSub(Xsp, P) then ¬ PSub(Xsp, P)	2.8
	∀ XP PSub(P, XP) then ¬ PSub(P, XP)	2.8
	∀ XD Dom(P, XD) then ¬ Dom(P, XD)	2.9
	∀ XR Rng(P, XR) then ¬ Rng(P, XR)	2.10
	∀ Xi, Yi PI(Xi, Yi, P) then ¬ PI(Xi, Yi, P)	2.14
¬ Indiv(I)	∀ XC CI(I, XC) then ¬ CI(I, XC)	2.11
	∀ Xi, XP PI(I, Xi, XP) then ¬ PI(I, Xi, XP)	2.14
	∀ Xi, XP PI(Xi,I, XP) then ¬ PI(Xi,I, XP)	2.14
¬ CSub(Xsc,XC)	if XC = Resource then ¬ Cl(Xsc)	2.12
	NON deterministic SITUATIONS:	
	1) ∀ Y such that CSub(Xsc, Y) and CSub(Y,XC) then	
	Choice (¬ CSub(Xsc,Y), ¬ CSub(Y, XC), both) or exception	2.18
	2) ∀ 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	2.21
	3) ∀ 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	2.23
¬ PSub(Xsp,XP)	NON deterministic SITUATION:	
	∀ Y such that PSub(Xsp, Y) and PSub(Y,XP) then	
	Choice (¬ PSub(Xsp,Y), ¬ PSub(Y, XP), both) or exception	2.20
¬ Dom(Xp, XD)	¬ Pr(Xp)	2.15
¬ Rng(Xp, XR)	¬ Pr(Xp)	2.15
¬ CI(Xi,XC)	if XC = resource then ¬ Indiv(Xi)	2.13
	NON deterministic SITUATIONS:	
	1) ∀ YC such that CI(Xi, YC) and CSub(YC,XC) then	
	Choice (¬ CI(Xi,YC), ¬ CSub(YC, XC), both) or exception	2.26
	2) ∀ Yi, Zp such that PI(Xi, Yi, Zp) and Dom(Zp,XC) then	
	Choice (¬ PI(Xi, Yi, Zp), ¬ Dom(Zp,XC), both) or exception	2.24
	3) ∀ Yi, Zp such that PI(Xi, Yi, Zp) and Rng(Zp,XC) then	
	Choice (¬ PI(Xi, Yi, Zp), ¬ Rng(Zp,XC), both) or exception	2.25
¬ PI(Xi,Yi, Xp)	NON deterministic SITUATIONS:	
	∀ Yp such that PI(Xi,Yi, Yp) and PSub(Yp,Xp) then	
	Choice (¬ PI(Xi,Yi, Yp), ¬ PSub(Yp,Xp), both) or exception	2.27

Table 5.2: UpdCond table for deletions.

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 2.26 in $\mathscr{C}$ (Fig. 2.3) 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. Non-deterministic situations are identified in blue in Table 5.2. Their management is further discussed in Sec. 5.2.

5.1.2 Enforcing pre-conditions and updates

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. This is discussed further in the next section.

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.

Intrinsically inconsistent updates could be problematic depending on the method chose to deal with side effects. Let's consider the following example.

Example 3 *Let U be an intrinsically inconsistent update requiring the insertion of a class instance $CI(Excipient, Excipient)$ in $\mathbb{G}$ of Fig 2.2 (rule r_{CI}). Following Algorithm 1, conditions $c1 : Ind(Excipient)$ and $c2 : Cl(Excipient)$ are obtained by FindPredCond2ApplyUpd. However, these two conditions are contradictory since they engender inconsistent update requests, namely: $Ind(Excipient)$ and $\neg Ind(Excipient)$ and also $Cl(Excipient)$ and $\neg Cl(Excipient)$.*

Obviously, according to the method chosen for dealing and ordering side-effects on line 2 of Algorithm 1, inconsistent updates may result in cycles. 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 and we can derive the following lemma:

Lemma 2 (SetUp Correction and termination) *Let $\mathbb{G}$ be a consistent graph and $\mathbb{R}$ our set of graph rewriting rules. Let U be an update, F the fact being inserted or deleted. Let $\mathscr{D}'/\mathbb{G}'$ be the database such that $\mathbb{G}' = \mathsf{SetUp}(\mathbb{G}, \mathbb{R}, U)$. Then,*

- *SetUp terminates and*
- *if U is not intrinsically inconsistent,*

– *if* U *is an insertion* $F \in \mathscr{D}'$
– *if* U *is a deletion* $F \notin \mathscr{D}'$.

□

Proof 2 *The proof is provided in Sec. 5.3 by considering all possible atomic updates.*

In conclusion, 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) $\mathbb{G}^1 = r_1(\mathbb{G})$, $\mathbb{G}^2 = r_2(\mathbb{G}^1)$, ... $\mathbb{G}^n = r_n(\mathbb{G}^{n-1})$ 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.

5.2 Handling non-determinism

As previously discussed, different ways of enforcing an update could lead to different databases due to:

1. the considered set of side-effects when several are acceptable (see situations identified in blue in Table. 5.2).
2. the order in which updates are applied.

In our current implementation, non-determinism is solved through arbitrary choices. A future version will integrate more advanced mechanisms as discussed bellow.

5.2.1 Order of updates

PreConditions can be seen as a set (updates treated on any order) or as a list ordered according to a particular method. The current implementation of SetUp uses an arbitrarily pre-defined order. 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. Considering the Example 3, SetUp behaves as follows.

Example 4 *Following Algorithm 1 and the order established in table* UpdCond*, conditions* $c1 : Ind(Excipient)$ *and* $c2 : Cl(Excipient)$ *are obtained by FindPredCond2ApplyUpd. They engender inconsistent update requests, namely:* $Ind(Excipient)$ *and* $\neg Ind(Excipient)$ *and also* $Cl(Excipient)$ *and* $\neg Cl(Excipient)$.

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.*

□

At the end, $\mathbb{G}$therefore contains $Cl(Excipient)$. Swapping the order of conditions $c1$ and $c2$ would have resulted in a graph containing $Ind(Excipient)$.

Order of updates is particularly problematic when considering intrinsically inconsistent updates. One first solution would therefore to detect such updates and reject them without generating side-effects. However, this order also impact the consistent updates. Consider for example the suppression of a property P which as a sub-property P' both having an instance from A to B. This update will trigger the suppression of 1) all of $P's$ instances as well as 2) all the sub-property relations it is involved in. If 1) is handled before 2), the instance of P' from A to B will be suppressed as side-effect since P' is a sub-property of P. Said instance is not suppressed if 2) is handled before 1).

Two ordered list of updates generating different databases can be seen as two different set of updates and can therefore be handled similarly, as discussed in the next subsection.

5.2.2 Multiple acceptable set of updates

When considering certain type of fact suppression, several set of side-effects could be acceptable, as identified in blue in Table. 5.2. For example, when suppressing an instance of a property P between two individuals A and B, for any sub-property P' of P, we could:

1. suppress $PSub(P', P)$
2. suppress $PI(A, B, P')$
3. suppress both

In the current version of the implementation, 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. When non-determinism is over two side-effects implying changes on the schema, the priority is to break the highest hierarchical link. In the aforementioned example, SetUp therefore suppress $PI(A, B, P')$.

While these criterion seem reasonable, arbitrary choices are seldom the best solution. A better solution would be to evaluate the impact each set of updates has on data quality and pick the most suitable depending on the context. Performing such evaluations is the role of the the quality module of the SENDUP software suite. The current implementation of SetUp rely on arbitrary choices

as the quality module is not currently available. Interfacing SetUp with such a module is investigated in a dedicated report [5], a proof of concept being provided. Integration will be done within Task 4.

5.3 SetUp correction and termination

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.

$\neg Cl(A)$ $A \neq Resource$:

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

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

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).

$\boldsymbol{\neg CI(A, B)}$ **:**

↳ $\forall X s.t. Dom(X, B)$ 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).

$\boldsymbol{Indiv(A)}$ **:**

↳ $\neg Cl(A)$

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

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

↳ $\neg Pr(A) \rightarrow \forall X, Y \neg Pi(X, Y, A)$

↳ $CI(A, Resource)$

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.)

$\neg Pr(A)$:

↳ $\forall X, Y \neg Pi(X, Y, A)$

Proof ok for this case.

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

$Pi(A, B, C)$:

↳ 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.

Chapter 6

Experimental evaluation

SetUp is implemented using Java and AGG (The Attributed Graph Grammar System) [35]. AGG is one of the most mature and cited development environment supporting the definition of typed graph rewriting systems [29]. It supports the SPO approach as well as its main extension: PAC, NAC, and GAC. The current version of *SetUp* provides a textual interface and offers different updating modes, according to the user level. The complexity of *GraphRewriter* essentially determines SetUp's complexity. This chapter experimentally investigates SetUp in various update scenarios, evaluating their execution time and the number of generated side-effects. SetUp [5] and a report detailing its implementation are available online.

6.1 Methodology

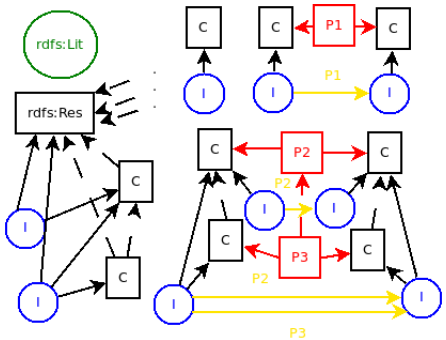


Figure 6.1: Experimental graph with $I = 1$ and $S = 2$

Datasets. The impact of the schema complexity (particularly, the complexity of the hierarchy set up in a schema) on the performance of our method is non-negligible. Thus, although there are many open RDF datasets available, our

Table 6.1: Experiment scenarios (C is a class, P a property and I an individual.)

Scenario		*Update type*												
Notation	*Explanation*	$\neg CI$	$\neg CL$	$\neg CSub$	$\neg PI$	$\neg Prop$	$\neg PSub$	$\neg Dom$	CI	CL	$CSub$	PI	$Prop$	$PSub$
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 exists P with the URI of C P is outside an hierarchy								✓					
$\exists P_{=CinH}$	there exists P with the URI of C P is at the top of a hierarchy								✓					

experiments are conducted on synthesised RDF/S graphs, allowing us to analyse results according to changes on the schema hierarchy. A simplified example is provided in Fig. 6.1 with the aforementioned graphical representation for typed edges and nodes. Experimental graphs are composed of: (A) Schema: (i) a minimal schema with no hierarchy (a property with two dom/rng classes and a class, illustrated in red and black in the upper right part of Fig. 6.1) plus (ii) a simple hierarchy of S classes and properties (illustrated in the bottom part of the figure). (B) Instances of all these classes and properties (in blue and yellow in the figure). Concepts outside or at the bottom of the hierarchy have I instances, the next has $2 * I$ instances, etc (so that the top of the hierarchy has S^*I instances). 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 6.1. 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 $\exists P_{=CinH}$).

Measurements. The time is measured with JMH [32] on 3 forks of 10 warmups and 50 measure iterations with a mean of 150 op./iteration.

6.2 Experimental results

Figs. 6.2 and 6.3 show the results of our experiments with regard to the number of generated side-effects and execution time, respectively.

6.2.1 Side-effects

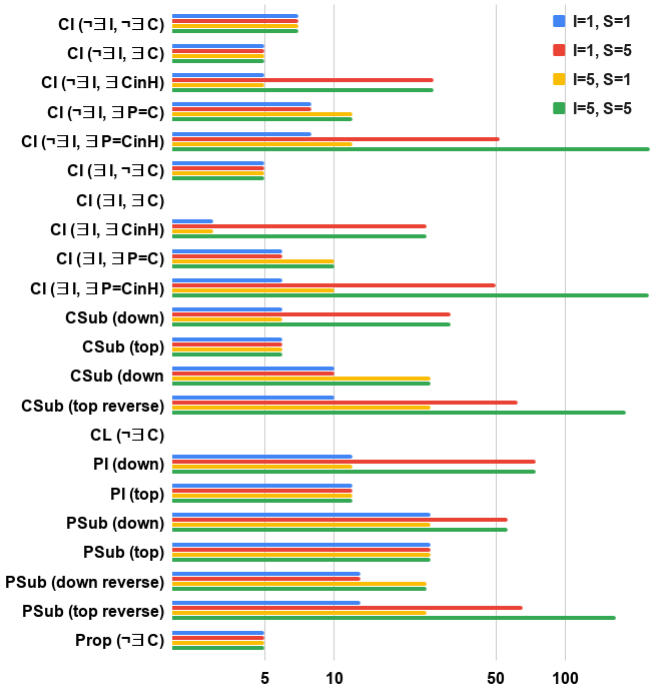
Side-effects tackled by the *GraphRewriter* are not taken into account: for instance, the deletion of a class at the top of the hierarchy is reported with 0 side-effect since deletion of relevant CI and $CSub$ are handled by the *GraphRewriter* through the removal of dangling edges. On the contrary, 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 $S * I$ $\neg PI$ (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 number of generated side-effects increases linearly with S and quadratically with I.

6.2.2 Execution time

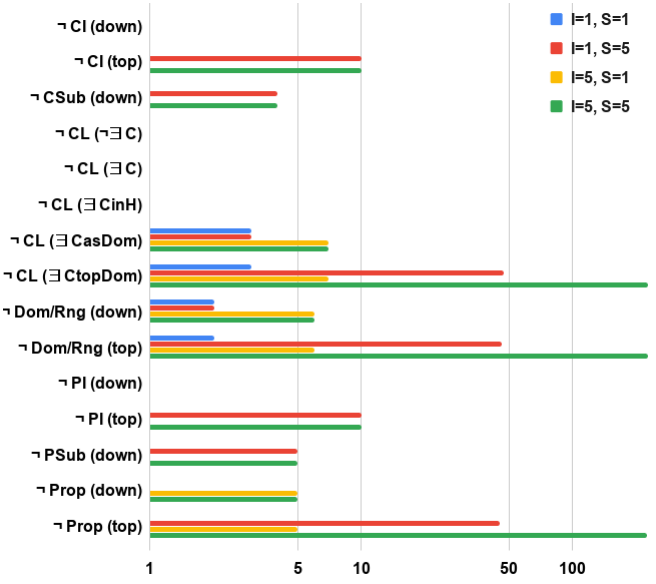
Execution time grows with the size of the knowledge graph, as it impacts the pattern matching and the verification of rule applicability phases. The scale of this impact varies depending on the complexity of the applied graph rewriting rules. $CL(A)$, for example, triggers the same number of side-effects by both SetUp and the *GraphRewriter* (which is $CSub(A, \text{“}rdfs : Resource\text{”})$) 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 $I = 5$

$(660, 5\text{s})$. By roughly tripling the size of the graph, each update $\neg CI$ 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 than $(5, 1)$ as it as almost as many nodes but twice as many edges. Yet, $\neg Dom/Rng$ (top) is almost three times longer on the second configuration ($190, 5$s and 538s, respectively). This is due to the number of side-effects going 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 $10, 3$s respectively with $I = S = 5$), even tough the latter implies the suppression of $S\ CSub$ relationships.

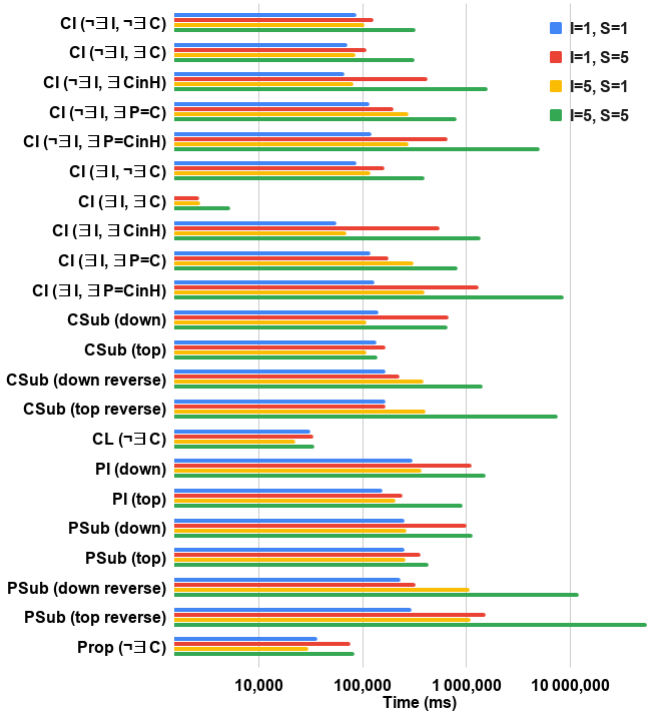


(a) Fact addition

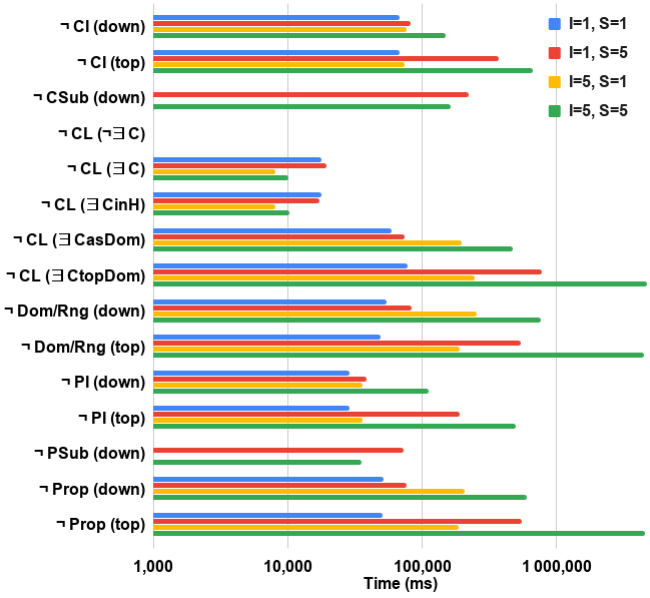


(b) Fact suppression

Figure 6.2: Number of side-effects.



(a) Time



(b) Side-effects

Figure 6.3: Experimental execution time.

Chapter 7

Related work

Consistent database updating has been considered in different contexts, always with two main goals: database evolution (by allowing changes) and constraint satisfaction (by keeping consistency *w.r.t.* the given rules). In this context, two aspects of our proposal can be considered as particularly original: (i) the use of graph rewriting techniques and (ii) the adoption of CWA with RDF data.

7.1 Graph rewriting for database updates

Concerning the first aspect, to generalize and abstract consistent updating methods, different works have used formalisms such as tree automata or grammars for XML ([28, 37] as surveys) or first order logic for relational (such as [38]) and, currently, graph databases (*e.g.* , [6, 8, 14]).

In spite of the importance of graphs in RDF and ontology representation, the use of formal graph rewriting techniques to model RDF evolutions is still mildly studied in this context. Formal graph rewriting techniques are usually based on *category theory*, an abstract way to deal with different algebraic mathematical structures (here, the graphs) and the relationships between them.

Algebraic approaches of graph rewriting allow a formal yet visual specification of rule-based systems characterizing both the effect of transformations and the contexts in which they may be applied. Studying the use of *graph* rewriting techniques to deal with *graph* models is the kernel of our motivation.

Few approaches relying on graph rewriting to formalize ontology evolutions have already been proposed [7, 31, 24]. They usually focus on formalization but do not provide an implementation.

To the best of our knowledge, only [23] proposes an implementation of an approach where graph rewriting is used to model ontology updates. Its objective is to tackle the evolution, alignment, and merging of OWL ontologies (see also [24]) with OWA under some consistency constraints. Nested and general application conditions are not considered in [23], thus, constraints relative to transitive properties are not tackled; their proposal cannot offer guarantees we

can (*e.g.* , the absence of cycles in subclass relationships).

7.2 CWA and OWA

Concerning the second aspect, since RDF data, in the web semantic world, is usually associated to the OWA, having CWA as the basis of our RDF database maintenance may be seen as atypical.

In SENDUP, the goal is to use RDF to represent connected data in a data-centered application. Even though we intend to present a general method which apply to any graph databases where consistency has to be preserved, our ultimate goal is to support the anonymisation process. We believe that adopting the CWA allows a better understanding and management of the published knowledge, which is crucial for anonymisation.

In this context it is worth mentioning, that work such as [4, 33, 36] brings back IC and CWA to the OWL world (sometimes through a hybrid approach), stressing the importance of our proposal.

7.3 Updating approaches

Now, to position our work in regards to other updating approaches, the following points deserve attention. Differences between update and revision are usually considered (we refer to [16] for an overview). These differences are the consequence of different views of the problem and influence the semantic of changes of each particular proposal. As in [6, 14], we consider updates as changes in the world rather than as a revision in our knowledge of the world ([16, 17]). In such update context, the chase procedure is usually associated to the generation of side-effects imposing extra insertions or deletions (*w.r.t.* those required by the user) to preserve consistency. Clearly, constraints are expected not only to be inherently consistent (*e.g.* , a set of constraints generating contradictory side effects for the same update u is not acceptable) but also to avoid contradicting the original intention of the user's update. The theory of consistency enforcement in databases has been the subject of various work, for instance [20, 21]. In our current approach, we only deal with RDF/S constraints whose consistency is ensured, but it could be extended to deal with user-defined constraints.

Several recent updating works focus on consistent graph databases. The approach in [25] differs from ours, by proposing a semantic measure based on the difference between original and updated RDF sub-graph. Both [6, 9] consider RDF updating methods, but the former goes deeper in the study of null values. A parallel can be done between saturation in [9], the chase in [6, 8, 14] and *SetUp*. Authors in [6, 8, 9, 14] offer home-made procedures to implement their methods: [9] deals only with the RDF instance constraints (Fig. 2.3); in [6, 8], constraints are user's tuple-generating-dependencies. Incomplete information and updates are the focus of [6, 14]. Schema evolution is mentioned in [8, 9].

More expressive constraints represent a barrier to the update determinism. This is tackled in [15] due to simple rules and in [8] due to a total ordering (which may be considered similar to the priority method in this paper).

Our RDF update strategy is different from proposals such as [3, 11] where constraints are just inference rules in OWA. Although some RDF technologies such as ShEx [34], SPIN [18], and SHACL [19] already take constraints into account, the originality of SetUp is in relying on well-studied *graph rewriting techniques* to ensure database consistent evolution, providing a *useful and modern* application for these formal tools. SetUp represents a test-bed for new database applications on the basis of graph rewriting.

Chapter 8

Concluding Remarks

SetUp is a theoretical and applied framework for ensuring consistent evolution of RDF graphs. The importance of SetUp is in its originality of using graph rewriting techniques under the closed world assumption to set an updating system. We specified 19 graph rewriting rules formalizing atomic RDF/S updates whose application *necessarily* preserves constraints. If an update cannot be applied, SetUp may generate additional consistency preserving updates to ensure its applicability. Hence, any non-contradictory update may eventually be applied in a consistency-preserving manner.

8.1 Expected usage

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. Not only can SetUp be used as a test-bed for updating approaches but also for further database applications.

In particular, SetUp can be used for the two target scenarios of SENDUP where a separate entity triggers updates in SetUp to conform to a privacy model such as k-anonymity or differential privacy.

- offline RDF graph anonymization, where a snapshot of a RDF graph is anonymized and openly published. In this case, a transformation time of several hours is inconsequential.
- the sanitization of query's response. Since the response is the graph being modified, its size should be small w.r.t. the database, and the execution time should be acceptable.

8.2 Appropriateness w.r.t. SENDUP scenarios

The advantage of SetUp to conduct such operations is threefold.

Consistency and property preservation Even though the produced graph is ultimately perturbed and not a "real" database, constraint satisfaction and property preservation is paramount. Indeed, any inconsistency may give indication to potential attackers and therefore jeopardyze privacy. We believe that graph rewriting rules are appropriate to guarantee constraint and property preservation, as seen in this paper.

Update enforcement through the generation of side-effects Since requested updates are required to conform to the chosen anonymity model, it is important to eventually guarantee their applications. Hence, refusing an update is not acceptable in this context, justifying they need of side-effect management as handled by the proposed framework.

Closed World Assumpiton Even if most works related to RDF updates adopt the open world assumption, the closed world assumption adopted by SetUp allows a better understanding and management of the published knowledge, which is crucial for anonymisation. Note that this is not inconsistent with the primary advantages of *linked data*; once published, the database can be subject to inference rules or linked to other knowledge-bases. Indeed, privacy models (*e.g.* , approaches based on differential privacy) do not necessarily make hypothesis on the attackers' auxiliary knowledge. Rather, they only focus on the released data and privacy guarantees stand regardless of existing and accessible data related to the published data base.

Bibliography

[1] https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf/

[2] Abiteboul, S., Manolescu, I., Rigaux, P., Rousset, M.C., Senellart, P.: Web data management. Cambridge University Press (2011)

[3] Ahmeti, A., Calvanese, D., Polleres, A.: Updating RDFS ABoxes and TBoxes in SPARQL. CoRR **abs/1403.7248** (2014)

[4] Cerans, K., Barzdins, G., Liepins, R., Ovcinnikova, J., Rikacovs, S., Sprogis, A.: Graphical schema editing for stardog OWL/RDF databases using OWLGrEd/S **849** (01 2012)

[5] Chabin, J., Eichler, C., Halfed Ferrari, M., Hiot, N.: SetUp: a tool for consistent updates of rdf knowledge graphs, {Online,}{https://www.univ-orleans.fr/lifo/evenements/sendup-project/index.php/softwares/setup-schema-evolution-through-updates}

[6] Chabin, J., Halfeld Ferrari, M., Laurent, D.: Consistent updating of databases with marked nulls. Knowledge and Information Systems (2019)

[7] De Leenheer, P., Mens, T.: Using graph transformation to support collaborative ontology evolution. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) Applications of Graph Transformations with Industrial Relevance. pp. 44–58. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

[8] Flouris, G., Konstantinidis, G., Antoniou, G., Christophides, V.: Formal foundations for RDF/S KB evolution. Knowl. Inf. Syst. **35**(1), 153–191 (2013)

[9] 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)

[10] Golas, U., Biermann, E., Ehrig, H., Ermel, C.: A visual interpreter semantics for statecharts based on amalgamated graph transformation. ECEASST **39** (01 2011)

[11] Gutierrez, C., Hurtado, C.A., Vaisman, A.A.: RDFS update: From theory to practice. In: The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC, Greece, Proceedings, Part II. pp. 93–107 (2011)

[12] 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`

[13] 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)

[14] 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)

[15] 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)

[16] Hansson, S.O.: Logic of belief revision. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, winter 2016 edn. (2016)

[17] Katsuno, H., Mendelzon, A.O.: On the difference between updating a knowledge base and revising it. In: Proc. of the 2nd Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25. pp. 387–394 (1991)

[18] Knublauch, H., Hendler, J.A., Idehen, K.: SPIN - overview and motivation. W3C member submission. http://www.w3.org/Submission/2011/SUBM-spin-overview-20110222 (2011)

[19] Knublauch, H., Ryman, A.: Shapes constraint language (SHACL). W3C first public working draft, w3c. http://www.w3.org/TR/2015/WD-shacl-20151008/. (2017)

[20] Link, S.: Towards a tailored theory of consistency enforcement in databases. In: Foundations of Information and Knowledge Systems, Second International Symposium, FoIKS, Germany, Proceedings. pp. 160–177 (2002)

[21] Link, S., Schewe, K.: An arithmetic theory of consistency enforcement. Acta Cybern. **15**(3), 379–416 (2002)

[22] Löe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science **109**(12), 181 – 224 (1993)

[23] 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)

[24] 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`

[25] 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)

[26] 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)

[27] 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)

[28] Schwentick, T.: Automata for XML - A survey. J. Comput. Syst. Sci. **73**(3), 289–315 (2007)

[29] 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)

[30] Serfiotis, G., Koffina, I., Christophides, V., Tannen, V.: Containment and minimization of RDF/S query patterns. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6-10, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3729, pp. 607–623. Springer (2005)

[31] Shaban-Nejad, A., Haarslev, V.: Managing changes in distributed biomedical ontologies using hierarchical distributed graph transformation. International Journal of Data Mining and Bioinformatics **11**(1), 53–83 (2015)

[32] Shipilev, A., Kuksenko, S., Astrand, A., Friberg, S., Loef, H.: OpenJDK Code Tools: JMH (2007), `https://openjdk.java.net/projects/code-tools/jmh/`

[33] Sirin, E., Smith, M., Wallace, E.: Opening, closing worlds - on integrity constraints. In: Dolbear, C., Ruttenberg, A., Sattler, U. (eds.) Proceedings

of the Fifth OWLED Workshop on OWL: Experiences and Directions, collocated with the 7th International Semantic Web Conference (ISWC-2008). CEUR Workshop Proceedings, vol. 432. CEUR-WS.org (2008)

[34] Solbrig, H., hommeaux, E.P.: Shape expressions 1.0 definition. W3C member submission. http://www.w3.org/Submission/2014/SUBM-shex-defn-20140602 (2014)

[35] Taentzer, G.: Agg: A graph transformation environment for modeling and validation of software. In: AGTIVE (2003)

[36] Tao, J., Sirin, E., Bao, J., McGuinness, D.L.: Integrity constraints in OWL. In: Fox, M., Poole, D. (eds.) Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, , USA. AAAI Press (2010)

[37] Tekli, J., Chbeir, R., Traina, A.J.M., Jr., C.T.: XML document-grammar comparison: related problems and applications. Central Eur. J. Comput. Sci. **1**(1), 117–136 (2011)

[38] Winslett, M.: Updating Logical Databases. Cambridge University Press, New York, NY, USA (1990)