Graph rewriting rules for RDF database evolution: optimizing side-effect processing

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

Graph Rewriting Rules for RDF Database Evolution: Optimizing Side-Effect Processing

Jacques Chabin[1], Cédric Eichler[2], Mirian Halfeld-Ferrari[1], and Nicolas Hiot[1,3]

[1] *Université d'Orléans, INSA CVL, LIFO EA, 45100 Orléans, France*
mail: {jacques.chabin,mirian}@univ-orleans.fr, nicolas.hiot@etu.univ-orleans.fr
[2] *INSA CVL, Université d'Orléans, LIFO EA, 18022 Bourges, France*
mail: cedric.eichler@insa-cvl.fr
[3] *Ennov Paris (Siège), 149 avenue de France, 75013 Paris, France*

Abstract

Purpose. Graph rewriting concerns the technique of transforming a graph; it is thus natural to conceive its application in the evolution of graph databases. The paper proposes a two-step framework where (i) rewriting rules formalize instance or schema changes, ensuring graph's consistency with respect to constraints, and (ii) updates are managed by ensuring rule applicability through the generation of side-effects: new updates which guarantee that rule application conditions hold.
Design/methodology/approach. The paper proposes $\mathsf{SetUp}^{\text{ND}}_{\mathsf{opt}}$, a theoretical and applied framework for the management of RDF/S database evolution on the basis of *graph rewriting rules*. The framework is an improvement of SetUp which 1) avoids the computation of superfluous side-effects and 2) proposes, via $\mathsf{SetUp}^{\text{ND}}_{\mathsf{opt}}$, a flexible and extensible package of solutions to deal with non-determinism.
Findings. The paper shows graph rewriting into a practical and useful application which ensures consistent evolution of RDF databases. It introduces an optimised approach for dealing with side-effects and a flexible and customizable way of dealing with non-determinism. Experimental evaluation of $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$ demonstrates the importance of the proposed optimisations as they significantly reduce side-effect generation and limit data degradation.
Originality. $\mathsf{SetUp}_{\mathsf{opt}}$ originality lies in the use of graph rewriting techniques under the closed world assumption to set an updating system which preserves database consistency. Efficiency is ensured by avoiding the generation of superfluous side-effects. Flexibility is guaranteed by offering different solutions for non-determinism and allowing the integration of customized choice functions.

1 Introduction

This paper focus on the application of graph rewriting in the evolution of graph databases. It shows the utility of this formal tool into a practical and useful application, by proposing a framework which ensures the *consistent*, *efficient*, and *flexible* evolution of RDF (Resource Description Framework) databases. Being a graph database, RDF management inspires the use of graph oriented tools. Initially just a part of the semantic web stack, RDF is currently largely used for representing high-quality connected data. 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 this paper, we consider databases satisfying integrity constraints (IC) and the Closed World Assumption (CWA) semantics. Indeed, the OWA is not

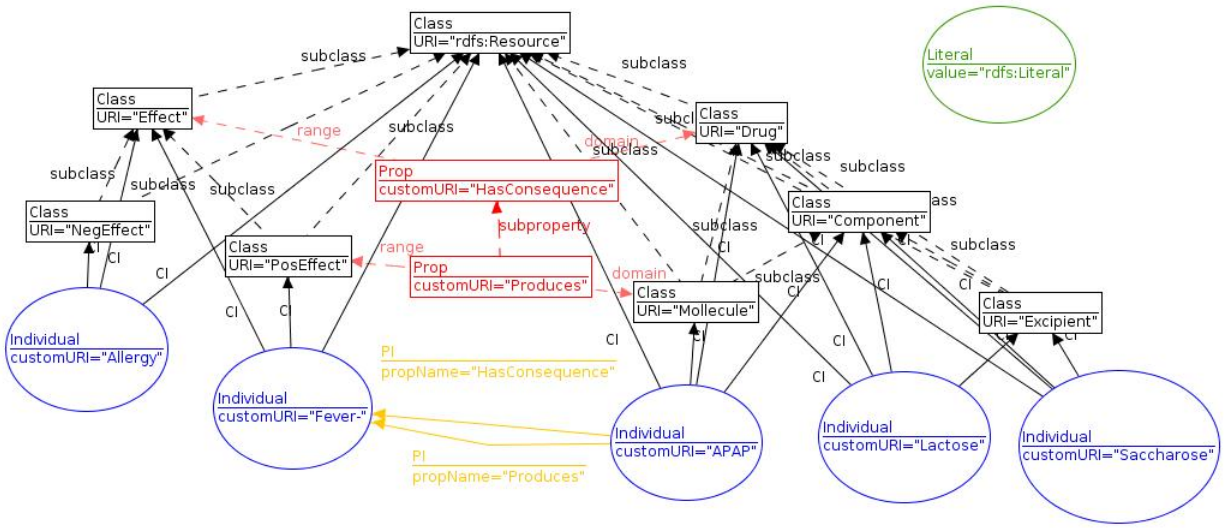

Figure 1: RDF schema and instance as a typed graph.

adapted to data-centric applications needing complete and valid knowledge Tao et al. (2010). 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. Currently working in the pharmacology domain, the following example illustrates our motivation.

Example 1 (Motivating Example.) *Fig. 1 shows a complete RDF/S graph database consistent w.r.t. 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 also be 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 be the same or sub-classes of those of HasConsequence?* □

This paper proposes $\mathsf{SetUp}_{\mathsf{opt}}$ (Schema Evolution Through UPdates, optimized version) 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. $\mathsf{SetUp}_{\mathsf{opt}}$ is built on SetUp, introduced in Chabin et al. (2020b), by avoiding some superfluous side effect computation and by proposing flexible solutions for non-determinism. $\mathsf{SetUp}_{\mathsf{opt}}$ ensures sustainability since it offers the capability of *efficiently* dealing with evolution of data instance and structure without violating the semantics of the RDF model.

$\mathsf{SetUp}_{\mathsf{opt}}$ summarized in three main steps

(1) Firstly we formalize atomic updates as *graph rewriting rules* encompassing integrity constraints: An *Update* is a general term and can be classified through two different aspects: whether it concerns the insertion or the suppression of a fact on one hand, and whether it concerns the instance or the schema on the other. Each atomic update is formalized by a graph rewriting rule whose application *necessarily* preserves the database 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, we provide procedure to enforce the (valid) application of an update: 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*. These side-effects are new updates that

should be performed to allow the satisfaction of a rule's conditions. Side-effects are implemented by procedures associated to an update type, and thus, to some rewriting rule. When an evolution is mandatory, we enforce database evolution by performing *side-effects* (*i.e.*, triggering other updates or schema modifications which will render possible rule application).
(3) $\mathsf{SetUp}_{\mathsf{opt}}$ shares steps (1) and (2) with SetUp but goes further by avoiding redundant side-effect generation on step (2) and by proposing different possible solutions for non-determinism with $\mathsf{SetUp}^{\text{ND}}_{\mathsf{opt}}$. Indeed, step (2) may present a choice among the new updates to be performed in order to allow rule application. SetUp dealt with this problem by imposing arbitrary choices where updates on instances are preferred to updates on schema. $\mathsf{SetUp}_{\mathsf{opt}}$ is presented in a new flavour, $\mathsf{SetUp}^{\text{ND}}_{\mathsf{opt}}$, where more sophisticated and flexible politics are implemented through a modular and customizable choice function.

Paper Organization. After some related work in Section 2, Section 3 sets up the work context and vocabulary used throughout this paper. Section 4 introduces the background on graph rewriting and provides an example of a graph rewriting rule formalizing an atomic update. Section 5 deals with side-effects and Section 6 discusses non-determinism. Section 7 provides an experimental evaluation of $\mathsf{SetUp}_{\mathsf{opt}}$ and $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$. Conclusions and perspectives are drawn in Section 8.

2 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. This section firstly discusses on these two aspects and then positions our paper in regards to other updating approaches.

Graph rewriting for database updates. To generalize and abstract consistent updating methods, different works have used formalisms such as tree automata or grammars for XML (Schwentick (2007) as a survey) or first order logic for relational (such as Winslett (1990)) and, currently, graph databases (*e.g.*, Chabin et al. (2019b); Flouris et al. (2013)). 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 De Leenheer and Mens (2008); Shaban-Nejad and Haarslev (2015); Mahfoudh et al. (2015). They usually focus on formalization but do not provide an implementation. To the best of our knowledge, only the work in Mahfoudh et al. (2015) is associated to an implementation where graph rewriting is used to model ontology updates. Nested and general application conditions are not considered in Mahfoudh et al. (2015), 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).

CWA and OWA. 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 this paper, the goal is to use RDF to represent connected data in a data-centered application. We intend to present a general method which applies to any graph databases where consistency has to be preserved. Our ultimate goal is to support the anonymisation process and 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 Cerans

• **Typing Constraints:**

$CL(x) \Rightarrow URI(x)$ (1) $Pr(x) \Rightarrow URI(x)$ (2) $Ind(x) \Rightarrow URI(x)$ (3)

$(CL(x) \wedge Pr(x)) \Rightarrow \bot$ (4) $(CL(x) \wedge Ind(x)) \Rightarrow \bot$ (5) $(Pr(x) \wedge Ind(x)) \Rightarrow \bot$ (6)

$CSub(x,y) \Rightarrow CL(x) \wedge CL(y) \vee y = Resource$ (7) $PSub(x,y) \Rightarrow Pr(x) \wedge Pr(y)$ (8)

$Dom(x,y) \Rightarrow Pr(x) \wedge CL(y) \vee y = Resource$ (9) $Rng(x,y) \Rightarrow Pr(x) \wedge CL(y) \vee y = Literal \vee y = Resource$ (10)

$CI(x,y) \Rightarrow Ind(x) \wedge CL(y) \vee y = Resource$ (11) $CL(x) \Rightarrow CSub(x, Resource)$ (12)

$Ind(x) \Rightarrow CI(x, Resource)$ (13) $PI(x,y,z) \Rightarrow Ind(x) \wedge (Ind(y) \vee Lit(y)) \wedge Pr(z)$ (14)

• **Schema Constraints:**

$Pr(x) \Rightarrow (\exists y,z)(Dom(x,y) \wedge Rng(x,y))$ (15) $((y \neq z) \wedge Dom(x,y) \wedge Dom(x,z)) \Rightarrow \bot$ (16)

$((y \neq z) \wedge Rng(x,y) \wedge Rng(x,z)) \Rightarrow \bot$ (17) $CSub(x,y) \wedge CSub(y,z) \Rightarrow CSub(x,z)$ (18)

$CSub(x,y) \wedge CSub(y,x) \Rightarrow \bot$ (19) $PSub(x,y) \wedge PSub(y,z) \Rightarrow PSub(x,z)$ (20)

$Psub(x,y) \wedge Dom(x,z) \wedge Dom(y,w) \wedge (z \neq w) \Rightarrow CSub(z,w)$ (21) $PSub(x,y) \wedge PSub(y,x) \Rightarrow \bot$ (22)

$Psub(x,y) \wedge Rng(x,z) \wedge Rng(y,w) \wedge (z \neq w) \Rightarrow CSub(z,w)$ (23)

• **Instance Constraints:**

$Dom(z,w) \Rightarrow (PI(x,y,z) \Rightarrow CI(x,w))$ (24) $Rng(z,w) \Rightarrow (PI(x,y,z) \Rightarrow CI(y,w) \vee (Lit(y) \wedge w = Literal))$ (25)

$CSub(y,z) \Rightarrow (CI(x,y) \Rightarrow CI(x,z))$ (26) $PSub(z,w) \Rightarrow (PI(x,y,z) \Rightarrow PI(x,y,w))$ (27)

Figure 2: Simplified and compacted form of RDF/S constraints

et al. (2012); Sirin et al. (2008); Tao et al. (2010) brings back IC and CWA to the OWL world (sometimes through a hybrid approach), stressing the importance of our proposal.

Positioning our paper *w.r.t.* to other updating approaches. As in Chabin et al. (2019b), we consider updates as changes in the world rather than as a revision in our knowledge of the world (Hansson (2016), as an overview for revision). 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. 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 Maillot et al. (2014) differs from ours, by proposing a semantic measure based on the difference between original and updated RDF sub-graph. Both Chabin et al. (2019b); Goasdoué et al. (2013) consider RDF updating methods, but the former goes deeper in the study of null values. A parallel can be done between saturation in Goasdoué et al. (2013), the chase in Chabin et al. (2019b); Flouris et al. (2013) and *SetUp*. Authors in Chabin et al. (2019b); Flouris et al. (2013); Goasdoué et al. (2013) offer home-made procedures to implement their methods: Goasdoué et al. (2013) deals only with the RDF instance constraints (Fig. 2); in Chabin et al. (2019b); Flouris et al. (2013), constraints are user's tuple-generating-dependencies. Incomplete information and updates are the focus of Chabin et al. (2019b). Schema evolution is mentioned in Flouris et al. (2013); Goasdoué et al. (2013). More expressive constraints represent a barrier to the update determinism. This is tackled in Halfeld Ferrari and Laurent (2017) due to simple rules and in Flouris et al. (2013) 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 Ahmeti et al. (2014); Gutierrez et al. (2011) where constraints are just inference rules in OWA. Although some RDF technologies such as ShEx, SPIN, and SHACL already take constraints into account, the originality of $\mathsf{SetUp}_{\mathsf{opt}}$ is in relying on well-studied *graph rewriting techniques* to ensure database consistent evolution, providing a *useful and modern* application for these formal tools. $\mathsf{SetUp}_{\mathsf{opt}}$ represents a test-bed for new database applications on the basis of graph rewriting.

3 RDF databases and updates

A collection of RDF statements intrinsically represents a typed attributed directed multi-graph. Constraints on RDF facts can be expressed in RDFS (Resource Description Framework Schema), the schema language of RDF. In Flouris et al. (2013) 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*) SchPred $= \{CL, Pr, CSub, PSub, Dom, Rng\}$, used to define the database schema, standing respectively for classes, properties, sub-classes, sub-properties, property domain and range, and (*ii*) InstPred $= \{CI, PI, Ind, Lit\}$, used to define the database instance, standing respectively for class and property instances, individuals and literals. 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.

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. $\mathbb{V}$ are nodes with type in $\{CL, Pr, Ind, Lit\}$ and $\mathbb{E}$ are edges having type in $\{Dom, Rng, PSub, CSub, CI, PI\}$. The notation $\mathscr{D}/\mathbb{G}$ designates these two formats of a database.* □

Fig. 1 shows an RDF instance and schema as a typed graph whose specifications are available in Chabin et al. (2020a). 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*. Class *"Resource"* symbolizes the root of an RDF class hierarchy. The instance is represented by individuals which are elements of a class (*e.g.*, $APAP$) and their relationships, (*e.g.*, *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, Resource)* are for the schema description and *Ind(Saccharose)* or *CI(Saccharose, Excipient)* are for the instance description.

Constraints presented in Flouris et al. (2013) are those in Fig. 2 which is borrowed from Halfeld Ferrari and Laurent (2017). We recall from Flouris et al. (2013) that these constraints capture the RDF/S semantics and the restrictions imposed by Serfiotis et al. (2005) 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. We are interested in database that satisfy all constraints in $\mathscr{C}$.

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

As already mentioned, this paper adopts the closed world assumption (CWA) where constraints are not just inferences - they impose data restrictions.

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. An update F is intrinsically inconsistent if $\nexists \mathscr{D}, F \in \mathscr{D} \wedge (\mathscr{D}, \mathscr{C})$. An update is consistent if it is not intrinsically inconsistent.* □

Updates can be classified according to the predicate of F, *i.e.*, the insertion (or the deletion) of a class, a class instance, a property, etc. For each update type, a rewriting rule r describes when and how to transform a graph database. SetUp relies on a set of 19 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 $\mathscr{C}$. 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 $\mathscr{C}$. The idea is: given $\mathscr{D}/\mathbb{G}$ for $(\mathscr{D}, \mathscr{C})$ and update U corresponding to rule $r \in \mathbb{R}$; if $\mathbb{G}$' is the result of applying r on $\mathbb{G}$ then our goal is to have $(\mathscr{D}', \mathscr{C})$ for $\mathscr{D}'/\mathbb{G}'$.

4 Graph rewriting for consistency maintenance

In our proposal, rewriting rules formalize both graph transformations and the context in which they may be applied. These rules may be *fully specified graphically*, enabling an easy-to-understand yet

formal graphical view of the graph transformation. To prevent the introduction of inconsistencies during updates, we 1) formally specify rules of $\mathbb{R}$ formalizing atomic $\mathbb{G}$ evolution and 2) prove that every rule in $\mathbb{R}$ ensures the preservation of every constraints in $\mathscr{C}$.

In our approach, each type of atomic update corresponds to one of the 19 rules in $\mathbb{R}$. 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 left-hand side (respectively, in their right-hand side). Consider for instance constraint (11): if $CI(A, B)$ is in $\mathscr{D}$, then $\mathscr{D}$ should also contain a class B and an individual A. Hence, the graph rewriting rule formalizing the insertion of $CI(A, B)$ is designed so that it is applicable only in a database respecting these conditions.

Clearly, in this paper, it is not possible to present each 19 rules of $\mathbb{R}$. The following presents the background on graph rewriting illustrated by a single rule of $\mathbb{R}$. All rules and proofs are available in Chabin et al. (2020a). We adopt the Single Push Out (SPO) formalism (Löe (1993)) to specify rewriting rules as well as several of its extensions to specify additional application conditions and restrict their applicability: *Negative Application Conditions* (NACs) (Habel et al. (1996)), *Positive Application Conditions* (PACs), and *General Application Conditions* (GACs) (Runge et al. (2012)).

Example 2 *Consider the graph database of Fig. 1 and assume node Allergy exists in* $\mathbb{G}$ *but is only connected to node Resource and not to nodes such as Effect. Consider the insertion of* $CI(Allergy, Effect)$*, i.e., we want to update* $\mathbb{G}$ *by inserting Allergy as an instance of class Effect. As the update is the insertion of a class instance, the rule to be considered is* r_{CI} *(Fig. 3).* □

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 – 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]

Fig. 3a formalizes the SPO core of r_{CI} rule: L has one class-typed node with an attribute URI whose value is B and one individual-typed node with URI A, while R has the same two nodes and a CI-typed edge from *Ind(A)* to *CL(B)*. By convention, an attribute value within quotation mark (e.g. *"NegEffect"*) is a fixed constant, while a value noted without quotation mark (e.g. A) is a variable whose value may be given as an input or assigned according to the context. The partial morphism from L to R is specified in the figure by tagging graph elements - nodes or edges - in its domain and range with a numerical value. An element with value i in L is part of the domain of m and its image by m is the graph element in R with the same value i. For instance, in Fig. 3a, the notation *1:* for the individuals on L and R indicates that they are mapped through m.

A graph rewriting rule $r = (\mathsf{L}, \mathsf{R}, m)$ is applicable to a graph G iff there exists a total morphism $\tilde{m}$ from L to G. The result of the application of r to G with regard to $\tilde{m}$ is the object of the push-out of the diagram composed by L, R, G, m, and $\tilde{m}$. Informally, the application of r to G with regard to $\tilde{m}$ consists in modifying 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 $\tilde{m}(\mathsf{L} \backslash Dom(m))$); (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.

Example 3 *In Fig. 3a, the rule is applicable to any graph containing a class node with a URI B and an individual node with an URI A. Its application consists in adding a class instance edge from the individual to the class. Assuming that A and B are given as input, this rule is thus a first way of formalizing the addition of a class instance relation. It is therefore the basis for including Allergy as an instance of Effects. However, this a naive rule: for instance, the node could already exist as an instance for the same class, creating a duplicate. To avoid this kind of situations, the rule applicability must be further restricted.* □

NACs and PACs are well-studied extensions that restrict rule application by, respectively, forbidding or requiring certain patterns in the graph. A NAC or a PAC for a rule r is defined as

[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.

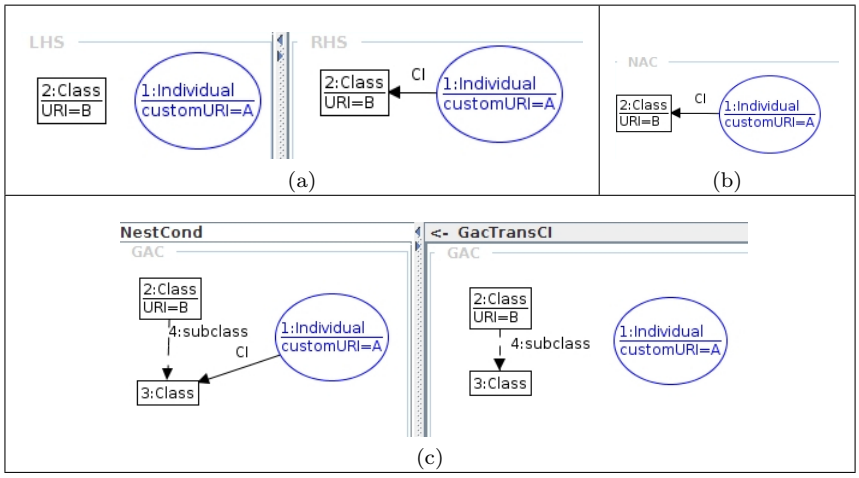

Figure 3: Insertion of a class instance: SPO core (a), NAC (b), and GAC (c).

a constraint graph which is a super-graph of its left-hand side. An SPO rule $r = (L, R, m)$ with NACs and PACs is applicable to a graph iff:
(i) there exists a total morphism $\tilde{m} : \mathsf{L} \to G$ (this is the classical SPO application condition);
(ii) for all PACs P (resp. NACs N) associated with r, there exists a total morphism (resp. there exists *no* total morphism) $\bar{m} : P \to G$ whose restriction to L is $\tilde{m}$.

By convention, since NACs and PACs are super-graphs of L unnecessary parts of L are not depicted when illustrating a NAC or a PAC. Graph elements that are common to L and the depicted part of the NAC are identified by a numerical value similarly to elements mapped by the morphism between L and R .

Example 4 *Fig. 3b specifies a NAC for the SPO rule in Fig. 3a. It forbids the application of the rule if $CI(A, B)$ already exists in the database but, it does not guarantee the satisfaction of propagation of class instances to super-classes.* □

GACs. 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. From this observation, *nested application conditions* Golas et al. (2011); Habel and Pennemann (2009) are introduced allowing the definition of conditions on the constraint graphs. A condition over a graph G is of the form *true* or $\exists(a, c)$ where $a : G \to C$ is a graph morphism from G to a condition graph C, and c is a condition over C. Now, a PAC P over a rule $(\mathsf{L}, \mathsf{R}, m)$ can be seen as a condition $(a, true)$, with a being the identity morphism from L to P while a NAC N can be seen as a condition $\neg(a, true)$ with a similar definition of a. GACs Runge et al. (2012) are a combination of nested application condition allowing the definition of complex application conditions for SPO rules. A GAC of a rule $(\mathsf{L}, \mathsf{R}, m)$ is a condition over L that may be quantified by $\forall$ and combined using $\wedge$ and $\vee$. The rule $(\mathsf{L}, \mathsf{R}, m)$ 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 $\dot{m} : G \to C$ such that $\dot{m} \circ a = \tilde{m}$ and $\dot{m}$ satisfies c.

Example 5 *Fig. 3c specifies a GAC of form $\forall(a, c)$ for r_{CI}. The morphism a from L to GacTransCI is depicted on the right part of Fig. 3c. 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. 3c): 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 $\dot{m}$ 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 $\dot{m}$.

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)). □

To prove that rule r_{CI}, defined in Fig. 3, preserves consistency, we consider the impacted constraints in $\mathscr{C}$, namely: 11 and 26 in Fig 2 (having atoms with CI on their L). The SPO part of r_{CI} ensures that the insertion of a class instance is performed only when the individual and its type already exist in the database (constraint 11). According to r_{CI}'s GAC, r_{CI} is applicable only if A is an instance of all super-classes of C (ensuring constraint 26). The correctness of all other rewriting rules is proved in a similar way in Chabin et al. (2020a). Based on that work we can also prove the following lemma.

Lemma 1 (Correctness of rewriting rules) *Let U be a consistent 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. $F \notin \mathscr{D}'$).* □

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, we generate 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 strategy.

Example 6 *Let $\mathscr{D}/\mathbb{G}$ be the database as the one in Fig. 1, but without NegEffect and its incident edges. Consider that U is the insertion of class instance $CI(Allergy, NegEffect)$ and $r_{CI} \in \mathbb{R}$ the corresponding rule (Fig. 3). The rule cannot be applied on $\mathbb{G}$ since it requires the existence of both the class and the individual which we want to "link together". If two side-effects are generated: (U^1) the insertion of an individual Allergy and (U^2) the insertion of class NegEffect, their corresponding rules are triggered, adding the individual and class and connecting them to class Resource. Rule r_{CI} can then applied, giving the graph of Fig. 1, except for a missing subclass edge between Effect and NegEffect and a missing CI edge from Allergy to Effect.* □

Roughly, the idea of SetUp is to allow 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. In Chabin et al. (2020b), the authors consider table UpdCond which is indexed by the update type and imposes a pre-established order to deal with the side-effects. Fig. 4(a) and 5(a) show an extract of UpdCond (*e.g.*, from the second row of the latter, we know that the insertion of $CI(A, B)$, depends on the existence of A as an individual, B as an class and the respect of hierarchical constraints). To design UpdCond for an insertion P, all constraints $c \in \mathscr{C}$ (Fig. 2) having atoms with the predicate of P in L_c (its body) are considered and updates corresponding to the atoms in R_c (its head) built. Deletions are treated in a reciprocal way, considering from the predicate of P on the heads of constraints and defining new updates based on the atoms in their bodies. It is worth noting that UpdCond is designed to contain all conditions needed for an update without taking into account any kind of *history* concerning previous applied updates. For instance, from Fig. 5(a), consider the insertion $CI(b, B)$. It engenders the insertion of $CL(B)$ even if we know – from a previous insertion in U – that B already exists as a class. The SetUp version (Chabin et al. (2020b)) takes into account all UpdCond, without any extra reasoning concerning the update 'status'.

In the current paper, we propose to optimize the SetUp version by proposing $\mathsf{SetUp}_{\mathsf{opt}}$. This new algorithm aims at avoiding unnecessary rule application and redundant side-effect generation. $\mathsf{SetUp}_{\mathsf{opt}}$ is based on a simple observation: when performing an update which is a side-effect of

a user's original update, the update-processing knows about operations previously performed and can be optimized on this basis. In other words, while we do not have any a priori knowledge on the original update U (we just know that $\mathbb{G}$ is consistent), we do have some new knowledge (concerning operations already performed) when dealing with U's side-effects. Thus, some constraint verification can be ignored at this second step. The following example illustrates the proposal.

Example 7 *We show how* $\mathsf{SetUp}_{\mathsf{opt}}$ *deals with the insertion of* $CI(I, A)$*. The first side effects ensure typing, with (1)* $Ind(I)$ *and (2)* $CL(A)$ *and schema constraints with (3)* $CI(I, B)$ *for each super-class* B *of* A*. Indeed (3) is generated by the application of rule 26, i.e.,* $CSub(y, z) \wedge CI(x, y) \Rightarrow CI(x, z)$ *with the instantiation* $x \rightarrow I$*,* $y \rightarrow A$ *and* $z \rightarrow B$*. Denote by* $U_1 = CI(I, B)$ *one update generated in (3). When dealing with* U_1*, we do not need to reconsider its side-effects. Indeed, we have:*
(4) $Ind(I)$ *which has already been treated in (1);*
(5) $CL(B)$ *which does not need to be checked, since the reason for having* U_1 *as* U*'s side-effect is that* B *is a class in* $\mathbb{G}$*;*
(6) $CI(I, C)$ *for each existing super-class* C *of* B*. These updates do not need to be generated. We have* $CSub(A, B)$ *and for each* C *we have* $CSub(B, C)$*. Then* $CSub(A, C)$ *is also true in* $\mathbb{G}$ *and* C*'s instances are already treated in (3).*
Therefore, in $\mathsf{SetUp}_{\mathsf{opt}}$*, any insertion* $U_1 = CI(I, B)$ *generated as a side-effect of an original insertion* $U = CI(I, A)$ *triggers no side-effect.* □

$\mathsf{SetUp}_{\mathsf{opt}}$ is summarized by Algorithm 1. 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 $\mathbb{G}$, $\mathsf{SetUp}_{\mathsf{opt}}$ computes, recursively, all updates necessary to change $\mathbb{G}$ into a new graph where r_U is applicable.

On line 1 of Algorithm 1, each condition c, necessary for applying r_U on $\mathbb{G}$, is added to *PreConditions*. Here, it is worth remarking the main differences between SetUp and $\mathsf{SetUp}_{\mathsf{opt}}$:

- In *FindPredCond2ApplyUpd*$_{\text{OPT}}$. While, in SetUp, *FindPredCond2ApplyUpd* works on table UpdCond for any update u, in $\mathsf{SetUp}_{\mathsf{opt}}$, *FindPredCond2ApplyUpd*$_{\text{OPT}}$ distinguishes between the original update U and its subsequent side effects u'. More precisely, we remark:
  - $\mathsf{SetUp}_{\mathsf{opt}}$ and SetUp treat original update in the same way,*i.e.*, by following UpdCond.
  - $\mathsf{SetUp}_{\mathsf{opt}}$ is built for insertions. Indeed the optimisations proposed in $\mathsf{SetUp}_{\mathsf{opt}}$ have no real impact for deletions since they do not required many useless verification. In fact, they often rely on dangling edges suppression which alleviate recursive calls and a priori analysis. Thus, for deletions, $\mathsf{SetUp}_{\mathsf{opt}}$ behaves as SetUp.
  - $\mathsf{SetUp}_{\mathsf{opt}}$ analyses the received update u according to its *history*. If u is a side-effect, $\mathsf{SetUp}_{\mathsf{opt}}$ activates an optimized side-effect table where superfluous verifications are avoided.
- $\mathsf{SetUp}_{\mathsf{opt}}$ requires parameter U_{prev} as input. This parameter indicates whether the update being treated is a original one (with an empty U_{prev}) or a side-effect (when U_{prev} in nonempty). Function *FindPredCond2ApplyUpd*$_{\text{OPT}}$ uses this information to launch the right treatment.

On line 2 of Algorithm 1, each condition c is considered. *PreConditions* can be seen as a set (updates treated on any order) or as a list ordered according to a particular method. For SetUp (in Chabin et al. (2020b)) a pre-defined order has been defined. $\mathsf{SetUp}_{\mathsf{opt}}$ is built on the optimisation of SetUp's pre-defined order. Indeed, when an update u' is a side-effect, its *history* may tell us that some verifications (normally imposed for an original update u) are not necessary, since they have already been done for one 'ancestor' of u'.

Once a condition c is chosen, *Planner2FitGraphInCond* (line 4) generates a new update set U' (*i.e.*, side-effects for U). Recursive calls (line 6) ensure that each side-effect $u' \in U'$ is treated. However, each call also sends the information concerning the 'father' of the update being treated; the result computed by function *FindPredCond2ApplyUpd* depends on the update *history*. When conditions for a rewriting rule $r_{u'}$ hold, function *GraphRewriter* applies $r_{u'}$ and the graph evolves.

Algorithm 1: $\mathsf{SetUp}_{\text{OPT}}(\mathbb{G}, \mathbb{R}, U, U_{prev})$

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

```
1: PreConditons := FindPredCond2ApplyUpd_OPT(G, R, U, U_prev)
2: for all condition c in PreConditons do
3:     if c is not satisfied in G then
4:         U' := Planner2FitGraphInCond (G, c)
5:         for all update u' in U' do
6:             G := SetUp_opt(G, R, u', U)
7: G := GraphRewriter(G, R, U)
8: return G
```

Eventually, if U is not intrinsically inconsistent, we obtain a new graph on which r_U is applicable (Chabin et al. (2020b)).

Correction of $\mathsf{SetUp}_{\mathsf{opt}}$

In Chabin et al. (2020b), the correction of SetUp is proved (proof in Chabin et al. (2020a), done on the basis of UpdCond). The correction of $\mathsf{SetUp}_{\mathsf{opt}}$ is established from the correction of SetUp to which we add the analysis of the abandoned side-effect verifications. In the rest of this section, we consider the insertion of $PSub$, CI and PI. The former is presented in details while the latter two are presented in a summarized way. In proving the correction of the insertion of $PSub$, we explain how this update is performed within $\mathsf{SetUp}_{\mathsf{opt}}$. The reasoning for the insertion of $CSub$ is similar, but it is not presented here due to the lack of space.

PSub. The only update that may lead to the generation of $PSub$-typed side-effects is the insertion of a $PSub$ relationship. Fig. 4(a) is an extract of UpdCond showing the side effects triggered by the insertion $PSub(A, B)$. These side effects ensure constraints shown in Fig. 2:
(1) $s1$ and $s2$ ensure that concerned properties exist (ensuring typing constraints such as 4 and 6);
(2) $s3$ and $s4$ ensure that containment between properties are reflected in their domains and ranges (rules 21 and 23);
(3) $s5$ ensures that A is not a sub-property of B, avoiding cycle in sub-property relationships as stated by rule 22;
(4) $s6$ et $s7$ generate new $PSub$ updates, respectively, for each super-class X of B and for each sub-class X of A (they correspond to applications of the transitive rule 20). The application of transitivity is detailed in the proof of Lemma 2.
(5) $s8$ generates instance updates as side-effects of schema changes to propagate property instances to super-properties.

Notice that when dealing with updates generated in step (4) above, there is no need to perform steps (1), (2) and (3). In other words, side-effects considered in (1), (2) and (3) have to be performed only to the original update $PSub$ and not for the updates it generates as side-effects. Algorithm $\mathsf{SetUp}_{\text{OPT}}$ implements this politics whose soundness is proven by the lemma 2 whose proof (available in the Appendix A) explains details of the reasoning adopted by $\mathsf{SetUp}_{\text{OPT}}$.

Indeed, Fig. 4(b) summarizes the action of $\mathsf{SetUp}_{\text{OPT}}$ which corresponds to our conclusions in in the proof of Lemma 2, precisely:

- For performing the *schema* change required by the insertion $PSub(A, B)$, only one situation need to be considered: the one dealing with insertions, such as $PSub(P_j, B)$, which are side-effects of the original $PSub(A, B)$.
- For performing the *instances* changes implied in the required insertion $PSub(A, B)$, side-effect $s8$ (rule 27) need to be considered – triggered by $PSub(A, P_j)$, $PSub(P_j, B)$ and $PSub(P_j, P_k)$, which are side-effects of the original $PSub(A, B)$

In this table, U_1 is a generated update, *i.e.*, a side-effect generated during the processing of the original update. The second column shows U_{prev}, the update triggering U_1 (*i.e.*, U_1's 'father'). The third column indicates the U_1's side-effect generated by $\mathsf{SetUp}_{\text{OPT}}$. Notice that, in the first line of Fig. 4(b) we generate new $PSub(X, Y)$ side effects (their 'father' is $PSub(X, B)$). To know the new side effects generated in this case, we use the last line where A is renamed into X.

(a)

Original update	Side-effects	#
$PSub(A, B)$	if B is not a property : $Pr(B, Ressource, Ressource)$	s1
	if A is not a property : $Pr(A, DB, RB)$ with DB domain of B and RB range of B	s2
	if $DA = DB$ then nothing else $CSub(DA, DB)$	s3
	if $RA = RB$ then nothing else if $RB = Literal$ then $\neg Pr(A), Pr(A, DA, Literal)$	
	else if $RA = Literal$ then $\neg Pr(B), Pr(B, DB, Literal)$ else $CSub(RA, RB)$	s4
	$\neg PSub(B, A)$	s5
	$\forall$ X such that $PSub(B, X)$ then $PSub(A, X)$	s6
	$\forall$ X such that $PSub(X, A)$ then $PSub(X, B)$	s7
	$\forall$ I, J such that PI(I, J, A) then PI(I, J ,B)	s8

(b)

U_1: side effect of U_{prev}	U_{prev}	Side-effects generated by U_1
$PSub(X, B)$	$PSub(A, B)$	$\forall Y s.t. PSub(B, Y) : PSub(X, Y)$
		$\forall (I, J) s.t. PI(I, J, X) : PI(I, J, B)$
$PSub(A, Y)$	$PSub(A, B)$ (original or SE)	$\forall (I, J) s.t. PI(I, J, A) : PI(I, J, Y)$

Figure 4: Handling $PSub$ as an original update or as a side effect. (a) An extract of UpdCond showing side effects for insertion $PSub$. (DA is domain of A, DB domain of B, RA range of A and RB range of B.) (b) Side-effect optimizations when $PSub$ insertion is itself a side-effect. Green lines indicate side-effects that are terminal ones in the insertion computation processing.

CI and PI. PI and CI as side-effects for, receptively, $PSub$ and $CSub$ are treated when dealing with the latter (*e.g.*, Fig. 4). The insertion of a property instance PI may be the side-effect of the insertion of another PI (a consequence of the application of rule 27).

Figure 5(a) is an extract of UpdCond with the side effects for the insertion of CI and $PI(i, j, P)$, aiming in the latter case the satisfaction of rules 14 (si), 24 (sii), 25 (siii), and 27 (siv).

We propose Fig. 5(b) as a summary of the updates that may lead to side-effects PI or CI. For instance, from the 2nd column of Fig. 5(b), we know that CI may be triggered as side-effects by (i) another CI, (ii) the insertion of an individual Ind, (iii) the insertion of a sub-class relationship, or (iv) the insertion of a property instance PI. The 3rd column of the table in Fig. 5(b) indicates the knowledge on which the $\mathsf{SetUp}_{\mathsf{opt}}$ politics is defined. Then, to show the adopted politics, $\mathsf{SetUp}_{\mathsf{opt}}$ actions are summarized in Fig. 5(c). Let us illustrate our reasoning with an example. In Fig 5(b) let $CI(i, A)$ be triggered by $CI(i, X)$ (2nd column). In this situation, we know that it has been triggered since $CSub(X, A)$ and that the previous treatement ensure $CI(i, Y)$ for all Y super-class of X (3rd column). Thus, in Fig 5(c), we indicate that, if the side-effect $CI(i, A)$ comes from the insertion of $CI(i, X)$, then no verification is needed (everything has being done during the insertion of $CI(i, X)$). Indeed, $\mathsf{SetUp}_{\mathsf{opt}}$ proceeds in the same way for the three first cases in the 2nd row of Fig 5(b), *i.e.*, there is nothing to be tested, as shown in the last line of Fig 5(c).

The situation is different when the $CI(i, A)$ is generated by a PI (Fig 5(c), top line of 2nd row). The following side-effects should be generated: $\forall B$ s.t. $CSub(A, B)$, $CI(i, B)$. Indeed, if $CI(i, A)$ is a consequence of the insertion of a property instance (PI), the instance i should be inserted as an instance of all super-classes B of A. Notice however that such an insertion, $CI(i, B)$, correspond to the case treated in the previous paragraph (*i.e.*, (Fig 5(c), bottom line of 2nd row). We then know that no other verification is needed, and this is why the action is marked in green in Fig 5(c).

(a)

Original update	Side-effects	#
PI(i,j,P)	Ind(i), Ind (j) $\vee$ Lit(j), Pr(P)	si
	Let Dom (P, PD) then CI(i, PD)	sii
	Let Rng (P, PR) then CI(j, PR) $\vee$ (Lit(j) $\wedge$ PR=Literal)	siii
	$\forall$ Q s.t. PSub (P, Q) then PI(i, j, Q)	siv
CI(Xi,XC)	Indiv (Xi); CL (XC) $\vee$ (XC = Ressource);	sv
	$\forall$ YC CSub (XC, YC) then CI(Xi, YC)	svi

(b)

U_1: side-effect of U_{prev}	U_{prev}	Relevant Knowledge
$PI(i,j,Q)$	$PI(i,j,P)$	$PSub(P,Q)$ and $\forall Y s.t. PSub(C,Y) : PI(i,j,Y)$
	$PSub(P,Q)$	$PI(i,j,P)$ and $\forall Y s.t. PSub(P,Y) : PI(i,j,Y)$
$CI(i,A)$	$CI(i,X)$	$CSub(X,A)$ and $\forall Y s.t. CSub(A,Y) : CI(i,Y)$
	$Ind(i)$	A = resource and $\forall Y, \neg CSub(A,Y)$
	$CSub(X,A)$	$CI(i,X)$ and $\forall Y s.t. CSub(A,Y) : CI(i,Y)$
	$PI(i,X,Y)$	$Dom(Y) = A$
	$PI(X,i,Y)$	$Rng(Y) = A$

(c)

U_1: side-effect of U_{prev}	U_{prev}	Side-effects generated by U_1
$PI(A,B,C)$	$PI(A,B,X)$ or $PSub(X,C)$	$\emptyset$
$CI(i,A)$	$PI(i,X,Y)$ or $PI(X,i,Y)$	$\forall B$ s.t. $CSub(A,B)$, $CI(i,B)$
	$CI(i,X)$; $Ind(i)$; $CSub(X,A)$	$\emptyset$

Figure 5: Handling instanciation as an original update or as a side effect. (a) An extract of UpdCond showing side effects for insertion PI and CI. (b) Relevant knowledge when CI or PI is a side-effect, depending on the triggering update. (c) Side-effect optimizations when CI or PI insertion is itself a side-effect. Green lines indicate side-effects that are terminal ones in the insertion computation processing.

6 Non determinism: impact on side effects

When dealing with update on database with constraints, non-determinism is a classical problem. For instance, in Fig. 1, to avoid re-inferring fact $CI(Allergy, Effect)$ after its deletion, we should delete $F_1 = CSub(NegEffect, Effect)$, or $F_2 = CI(Allergy, NegEffect)$ or both. Like SetUp, $\mathsf{SetUp}_{\mathsf{opt}}$ adopts *pre-established* solutions for dealing with such kinds of non-deterministic situations: changes on instances are preferred to changes on schema.

$\mathsf{SetUp}^{ND}_{\mathsf{opt}}$, summarized by Algorithm 2, is a *flexible extension* of $\mathsf{SetUp}_{\mathsf{opt}}$, allowing the implementation of different politics for dealing with non-determinism. $FindPredCond2ApplyUpd^{\text{ND}}_{\text{OPT}}$ (Algorithm 2, line 1) generates a set of ordered lists of pre-conditions: each list corresponds to a solution for the implementation of update U. This set is the input of function $ChoosePreCond$ which selects one of these lists, that will be recursively applied by the algorithm. $ChoosePreCond$ can be easily changed to implement different politics.

6.1 Identifying and managing non-determinism causes

$FindPredCond2ApplyUpd^{\text{ND}}_{\text{OPT}}$ operates on UpCond and the optimizations proposed by $\mathsf{SetUp}_{\text{OPT}}$. In the absence of non-deterministic situations, it behaves as its counterpart in Algorithm 1. The solutions that $FindPredCond2ApplyUpd^{\text{ND}}_{\text{OPT}}$ proposes for non-determinism depend on the detected kind of non-determinism (classified according to their causes). In our approach, we identify three possible causes of non-determinism: (i) different lists of side-effects are possible for one update; (ii) updates can be treated by the graph rewriter or as a pre-condition (*i.e.*, a side-effect) and (iii) different possible update order. Below we briefly consider them in the context of $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$.

Dealing with different possible lists of side-effects. Deletions $\neg CSub$, $\neg PSub$, $\neg CI$, $\neg PI$ are source of non-determinism since each of the involved atoms is the head (R_c) of a constraint

Algorithm 2: $\mathsf{SetUp}^{ND}_{\mathsf{opt}}(\mathbb{G}, \mathbb{R}, U, U_{prev})$

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

1: $PossiblesPreconditions := FindPredCond2ApplyUpd^{\text{ND}}_{\text{OPT}}(\mathbb{G}, U, U_{prev})$
2: $PreConditions := ChoosePreCond(PossiblesPrecondition)$
3: **for all** condition c in $PreConditons$ **do**
4: **if** c is not satisfied in $\mathbb{G}$ **then**
5: $U' :=Planner2FitGraphInCond\ (\mathbb{G}, c)$
6: **for all** update u' in U' **do**
7: $\mathbb{G} := \mathsf{SetUp}^{ND}_{\mathsf{opt}}(\mathbb{G}, \mathbb{R}, u', U)$
8: $\mathbb{G} := GraphRewriter(\mathbb{G}, \mathbb{R}, U)$
9: **return** $\mathbb{G}$

$c \in \mathscr{C}$ whose body (L_c) has more than one atom (Fig. 2). We recall (Section 5) that UpCond is built on the basis of the constraints in $\mathscr{C}$ and it indicates the situations where different side-effect choices are possible. $FindPredCond2ApplyUpd^{\text{ND}}_{\text{OPT}}$ checks whether the update u corresponds to one of the cases above and generates a list of preconditions for each possible choice.

Updates as pre-conditions or through graph rewriting. In $\mathsf{SetUp}_{\mathsf{opt}}$ some side-effects are directly handled by the graph-rewriter, either by *desideratum* (the side-effect and the original update must be conducted simultaneously, *e.g.*, the deletion of a property comes with the deletion of its range and domain) or by *convenience* (the side-effect corresponds to the suppression of dangling edges, *e.g.*, the deletion of a property implies the deletion of the edges connecting it to its sub-properties). The latter solution seems efficient but may lead to data degradation as illustrated in the following example.

Example 8 *For the update $U = \neg Pr(HasConsequence)$ over the database in Fig. 1, the only required pre-condition is $U_1 = \neg PI(APAP, Fever, HasConsequence)$. As HasConsequence is a super-property of Produces, to ensure U_1, we have choices: $U_2 = \neg PI(APAP, Fever, Produces)$, or $U_3 = \neg PSub(Produces, HasConsequence)$ or both. As SetUp_{opt} gives priority to instance changes, U_2 is applied as a side-effect. However, U_3 is also performed because of the dangling edges resulting from the deletion of the node HasConsequence. One can argue that SetUp_{opt} 'deletes too much'. $\mathsf{SetUp}^{ND}_{opt}$ gives the possibility to change this politics by performing U_3 as a side-effect (instead of automatically leaving it to be performed by the graph rewriter). As a consequence, U_2 does not need to be performed and the resulting database comprises one more fact than in the first scenario, $PI(APAP, Fever, Produces)$. To sum up, in this scenario $FindPredCond2ApplyUpd^{\text{ND}}_{\text{OPT}}$ produces a set with two lists $\{l_1, l_2\}$. In l_1, updates concerning the schema, such as U_3, comes first than updates concerning the instances, such as U_2 (as we explain in the following paragraph discussing update order). List l_2 does not contain updates concerning schema, such as U_2.* □

Side-effects of deletions $\neg Pr$, $\neg CL$, $\neg Ind$, $\neg Lit$ may be treated as side-effects or through the graph rewriting. $\mathsf{SetUp}_{\mathsf{opt}}$ offers the possibility to let $FindPredCond2ApplyUpd^{\text{ND}}_{\text{OPT}}$ treat those updates originally sent to the graph rewriter by *convenience*. The function generates such a possibility for specific side effects of deletions $\neg Pr$ (side effect $\neg PSub$) and $\neg CL$ (side effects $\neg CSub$ and $\neg CI$). Thus, for such cases, $FindPredCond2ApplyUpd^{\text{ND}}_{\text{OPT}}$ engenders two side-effect lists, each one giving rise to a different database instance. Deletions $\neg Ind$, $\neg Lit$ are left to the graph rewriter – changing the politics for them does not result in different database instances.

Update order. The order in which pre-conditions are treated may impact the resulting database. We distinguish two situations where this can happen. Firstly, for $\neg PR$ (or $\neg CL$) as in Example 8, if U_2 is treated before U_3 we delete $PI(APAP, Fever, Produces)$; otherwise we keep it. Secondly, if inconsistent updates are considered as input, for instance the insertion $U = CI(Allergy, Allergy)$

in database Fig. 1, may result in a database where *Allergy* is an individual or a class depending on the order updates $U_a = Ind(Allergy)$ and $U_b = CL(Allergy)$ are performed.

The function $FindPredCond2ApplyUpd^{\text{ND}}_{\text{OPT}}$ outputs a set of lists. To prepare each list, it generates a set of updates to be performed, and then order them according to an established politics. Several ordered lists may be generated from a single set if relevant. $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$ implements the ordered lists by imposing a partial order on the pre-conditions as explained below. When several ordering are possible, any representative is chosen as the orders will be equivalent. Notice that even if intrinsically inconsistent updates may generate several ordered lists, we do not detail them here because we plan to accept only consistent updates.

For insertions:

- Pre-conditions related to typing (rules 1 to 14) always come first. Those related to instance constraints (rules 24 to 27) come second, and schema constraints come last.
- Pre-conditions with quantifiers (*i.e.*, with the form "∀...", as s6 to s8 in Fig. 4(a)) are instantiated. Then, $PSub$ and PI (resp. $CSub$ and CI) are ordered to be consistent with the partial order composed by sub-property (resp. sub-class) relationships (*i.e.*, $CI(i, A)$ executed before $CI(i, B)$ implies $\neg CSub(A, B)$; $CSub(A, B)$ executed before $CSub(A, C)$ implies $\neg CSub(B, C)$).

For deletions:

- Pre-conditions related to schema comes first, then instances, and finally typing.
- Pre-conditions with the form "∀" (*e.g.* s6 to s8 in Fig. 4 (a)) are instantiated. Then, $PSub$ and PI (resp. $CSub$ and CI) are ordered to be consistent with the partial order composed by sub-property (resp. sub-class) relationships (*i.e.*$\neg CI(i, A)$ executed before $\neg CI(i, B)$ implies $\neg CSub(B, A)$; $\neg CSub(A, B)$ executed before $\neg CSub(A, C)$ implies $\neg CSub(C, B)$).

The construction of a list of updates that represents the choice of dealing with updates as pre-conditions instead as through graph rewriting is performed by placing deletions of $PSub$ (or $CSub$) before those of PI (or, resp. CI). The reverse would produce a list whose application result is identical to the one obtained when choosing to let the graph rewriter in charge of deleting $PSub$ (as discussed in the previous paragraph, particularly in Example 8).

6.2 Handling choices in a modular way

All versions SetUp, $\mathsf{SetUp}_{\mathsf{opt}}$ and $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$ are implemented in Java and rely on AGG – The Attributed Graph Grammar System (Taentzer (2003)), one of the most mature development environment supporting the definition and application of typed graph rewriting systems (Segura et al. (2008)). All versions provide a textual interface (TUI) and offer different updating modes, according to the user level. While $FindPredCond2ApplyUpd^{\text{ND}}_{\text{OPT}}$ is intrinsically integrated in the code, *ChoosePreCond* is modular and trivial to replace. Indeed, we define an interface with a unique method, *ChoosePreCond*, which receives a set of *ArrayList* of updates and renders an *ArrayList* of updates. This interface is used throughout the code by levering Java's implicit type casting. Thus, when using $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$, one may propose any implementation of the aforementioned interface for non-deterministic side-effect management. Therefore, any user may implement his own version of *ChoosePreCond* and integrate it seamlessly to our tool. We currently propose three different implementations:

1. $Choice_{ui}$. The simplest version is a user-interaction tool that integrates a TUI and allows the user to choose his favourite update list.
2. $Choice_{\mathsf{SetUp}}$. This version gives priority to schema preservation and minimizes the generation of pre-conditions, mimicking the politics adopted in SetUp (Chabin et al. (2020b)). In this version and referring to the cases in Section 6.1, the following remarks hold.

(i) When dealing with different lists of updates, the choice falls on a single update, preferably one concerning an instance.
(ii) The possibility of dealing with an update as a pre-condition instead as through graph rewriting is not taken into account. For instance, in Example 8, where the set $\{l_1, l_2\}$ is generated, the choice falls on l_2.
(iii) The order impacts only intrinsically inconsistent update (*i.e.*, if the system input is an intrinsically inconsistent update, the order in which side-effects are taken into account can result in different database instances).

3. $Choice_{opt}$. This version behaves similarly to $Choice_{\mathsf{SetUp}}$ except when dealing with the deletion of a property. In such a situation, the chosen update list is the one where the deletion of $PSub$ is done before the deletions of PI. Thus, in Example 8, with the set $\{l_1, l_2\}$, the choice falls on l_1. We believe $Choice_{opt}$ is a good candidate to preserve the quality of the database. However, we emphasize the biggest advantage of $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$: *any* user may easily implement and integrates his own choice function.

7 Experimental evaluation

This section experimentally investigates the proposed extensions. To allow comparison with SetUp we reproduce experiments proposed in Chabin et al. (2019a) using: (1) $\mathsf{SetUp}_{\mathsf{opt}}$ to quantify the optimization in term of generated side-effects, and (2) $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$ with $Choice_{opt}$ to highlight the benefits of this new choice function.

7.1 Experimental context

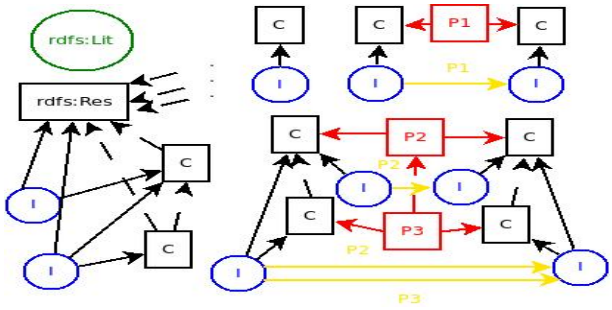

Figure 6: Experimental graph with $I = 1$ and $S = 2$

Datasets. For comparison sake, we use experimental datasets proposed in Chabin et al. (2019a). A simplified example is provided in Fig. 6 with the aforementioned graphical representation for typed edges and nodes. Experimental graphs are synthesized RDF/S graphs 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) plus (ii) a simple hierarchy of S classes and properties (illustrated in the bottom part of the figure). In what follows we note C_i (resp. P_i) the class (resp. property) of depth i within the hierarchy, so that C_1 is on top and C_S at the bottom. (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. Experimental scenarios consist in insertions and deletions categorized according to update type and database configurations, since the impact of an update is intrinsically related to these two factors. Considered scenarios are a sub-sets of those of Chabin et al. (2019a):

1. $\neg Prop(top)$: suppression of P_1.

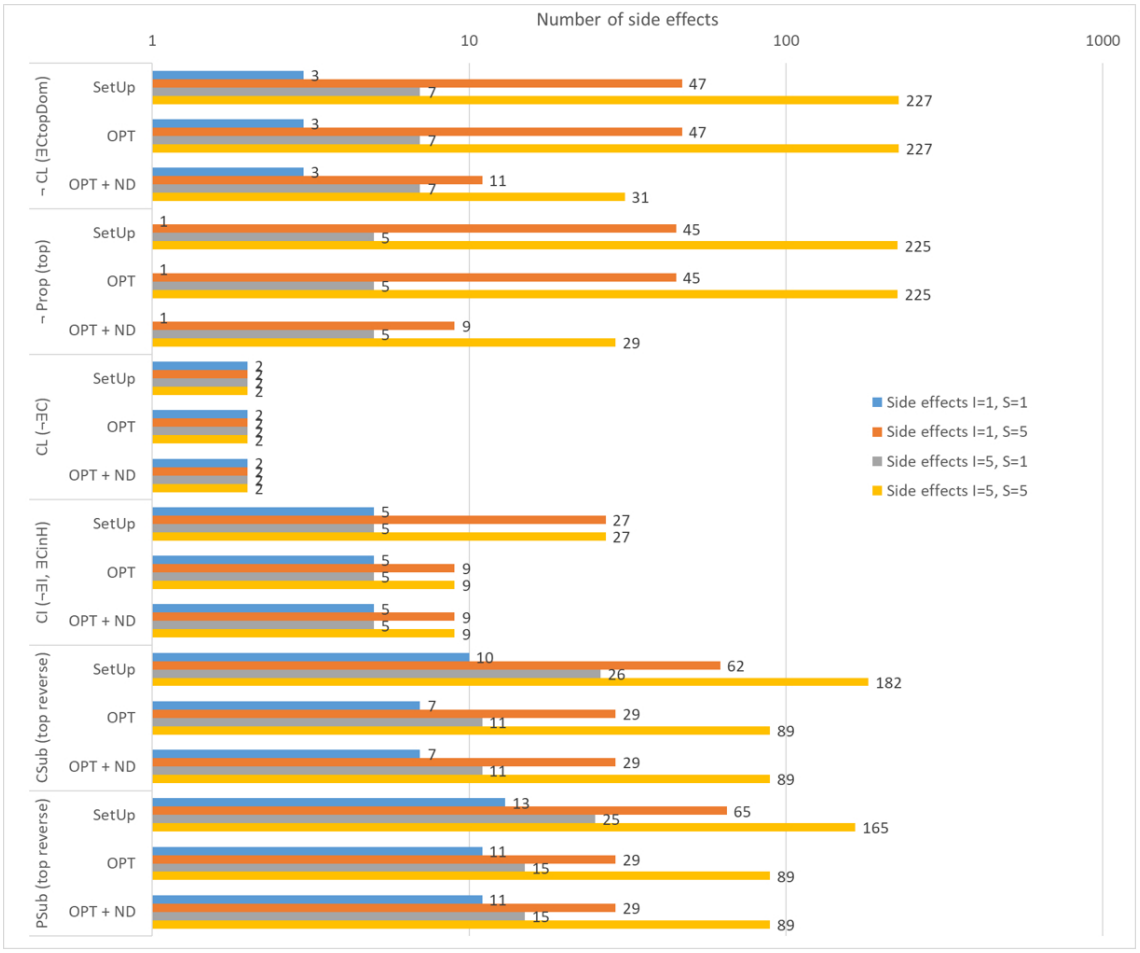

Figure 7: Number of side-effects generated by SetUp , $\mathsf{SetUp}_{\mathsf{opt}}$, and $\mathsf{SetUp}_{\mathsf{opt}}^{ND}$ using $Choice_{OPT}$.

2. $\neg CL(\exists CTopDom)$: suppression of a class C such that $Dom(P_1, C)$.
3. $CL(\neg\exists C)$: addition of a class C outside of the hierarchy.
4. $CI(\neg\exists I, \exists CinH)$: addition of $CI(ind, C_1)$ where $Ind(ind)$ is not in the database.
5. $CSub$ (resp. $PSub$) (top reverse) addition of $CSub(C_1, C)$ (resp. $PSub(P_1, P)$ where $CL(C)$ (resp. $Pr(P)$) is not in the database.

We chose to reproduce these scenarios as SetUp leads to an explosion of side-effects when suppressing P_1 (cases 1. and 2.) and when adding an entity on top of the hierarchy (cases 5.).
Experimental results. Figs. 7 shows the number of side-effects generated in each scenario using SetUp, $\mathsf{SetUp}_{\mathsf{opt}}$, and $\mathsf{SetUp}_{\mathsf{opt}}^{ND}$ with $Choice_{OPT}$. *Side-effects* are reported as the number of explicit preconditions and verifications; those tackled by the *GraphRewriter* are not taken into account.

7.2 Result interpretation

$\mathsf{SetUp}_{\mathsf{opt}}$ optimization. As expected, $\mathsf{SetUp}_{\mathsf{opt}}$ provides the same results as SetUp regarding deletions, but also insertions in simple scenarios (*e.g.*$CL(\neg\exists C)$). $\mathsf{SetUp}_{\mathsf{opt}}$ provides significant optimizations for insertions triggering multiples side-effects. For example, with $CI(\neg\exists I, \exists CinH)$ and S=5, $\mathsf{SetUp}_{\mathsf{opt}}$ generates 9 side-effects instead of 27, due to better CI propagation management.

In particular, it greatly improves the issue of inserting an entity on the top of the hierarchy. With $CSub$ (top reverse) and (I, S) =(1, 1), (1, 5), (5, 1), and (5, 5), SetUp produces 10, 62, 26, and 182 side effects, while $\mathsf{SetUp}_{\mathsf{opt}}$ only produces 7, 29, 11, and 89, respectively. A first explanation is that both tools generate initially $CI(i, C)$ for all the $S * I$ individuals i such that $CI(i, C_1)$. In $\mathsf{SetUp}_{\mathsf{opt}}$, these side-effects are terminal, while in SetUp each of these in turn generate typing checks ($Ind(i)$ and $CL(C_1)$) and $CI(i, Resource)$. Thus, with $S = 1$, SetUp generates $3 * I$ more side-effects than $\mathsf{SetUp}_{\mathsf{opt}}$. With $S \neq 1$, $\mathsf{SetUp}_{\mathsf{opt}}$ integrates even more optimizations and the difference is not only strictly more than $3^*I * S$, but also increases with S. Noticeably, the ratio of side-effect generated by SetUp to those generated by $\mathsf{SetUp}_{\mathsf{opt}}$ roughly goes from $1, 4$ to $2, 2$ as (I, S) goes from $(1, 1)$ to $(5, 5)$. Note that the size of the graph is a constant plus $2^*(S+S * I)$, meaning that the lower bound of redundant side-effects ignored by $\mathsf{SetUp}_{\mathsf{opt}}$ ($> I * S$) is roughly the size of the graph. This obviously depends on the database configuration; in general we can say that it is at least linear in the size of the part of the database impacted by the update.

Impact of choice: limiting data-degradation. The most problematic scenarios for SetUp (Chabin et al. (2019a)) are those leading to the suppression of a property on top of the hierarchy. Indeed, not only does it lead to an explosion of side-effect ($\neg Prop$ (top) leads to 45 and 225 side-effects with $(I, S) = (1, 5)$ and $(5, 5)$, respectively) but it also significantly degrades the database as all PIs are suppressed. The choice function $Choice_{opt}$ has been introduced to tackle the issue, as discussed in the previous section. Figure 7 shows that it only triggers 9 and 29 side-effects with $(I, S) = (1, 5)$ and $(5, 5)$, respectively. This is optimal w.r.t. the number of side-effect since it corresponds to $S - 1$ $\neg PSub$ (one per property in the hierarchy, except for the one been deleted) and $I * S$ $\neg PI$ (one per instance of the property been deleted). Furthermore, this solution is also optimal with regard to the minimization of deleted facts.

This shows the significance of $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$ firstly, arbitrary choices hard-coded in SetUp were sub-optimal w.r.t. the original aim, data conservation. Secondly, other objectives or metrics could dictate these choices, hence the importance of proposing a modular and easy way to propose new choice functions. This also illustrates the relevance of considering the possibility of tackling updates as pre-conditions rather than to delegate their application to the graph rewriter.

8 Conclusions and Perspectives

This paper proposes two major improvements of SetUp , a theoretical and applied framework for ensuring consistent evolution of RDF graphs whose originality lies in the use of a typed graph rewriting system. Each atomic update is formalized using a graph rewriting rule whose application *necessarily* preserves constraints. If an update cannot be applied, SetUp generate additional consistency preserving updates called side-effects to ensure its applicability.

Both improvement presented herein, $\mathsf{SetUp}_{\mathsf{opt}}$ and $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$ concern the management of these side-effects. $\mathsf{SetUp}_{\mathsf{opt}}$ avoids the generation of unnecessary verifications while recursively handling side-effects. This is done by leveraging knowledge of updates' history. Indeed, while we do not have any a priori knowledge on an original update U, we do have some new knowledge when dealing with U's side-effects. $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$ is a module of $\mathsf{SetUp}_{\mathsf{opt}}$ proposing a flexible and modular way of handling non-determinism. $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$ generates all different ways of guaranteeing the applicability of an update. The corresponding set of ordered lists of side-effects is transmitted to an easily customizable choice function. This function selects one of these lists and returns it to $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$ which ensures its recursive application. To demonstrate modularity and evolution capability, we implemented three different choice functions, including one taking user's input through a TUI.

The evaluation of SetUp exhibited performances issues with regard to two families of scenarios leading to an explosion of generated side-effects: the addition of an entity or the suppression of a property on top of a hierarchy. We show experimentally that both are addressed by our proposal. Indeed, $\mathsf{SetUp}_{\mathsf{opt}}$ greatly improves the first situations, ignoring a number of unnecessary side-effects roughly linear in the size of the experimental database. Regarding the second kind of scenarios, function $Choice_{opt}$ used alongside $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$ not only minimizes generated side-effects but also data

degradation. In a graph of size $(|V|, |E|) = (116, 480)$, the generated side-effects goes down from 225 to 29, each remaining side-effect being necessary and leading to the suppression of a fact.

We believe that the ameliorations proposed in this paper greatly improve the framework usability and wish to further investigate metrics to make decisions in case of non-determinism. In particular, we plan to use $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$ on offline RDF graph anonymization, where a snapshot of a RDF graph is anonymized and openly published. In this context, a separate entity triggers updates in $\mathsf{SetUp}^{ND}_{\mathsf{opt}}$ to conform to a privacy model such as k-anonymity or differential privacy. In this case, decisions should consider utility but also take into account published knowledge to preserve privacy.

References

S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, and P. Senellart. *Web data management*. Cambridge University Press, 2011.

A. Ahmeti, D. Calvanese, and A. Polleres. Updating RDFS ABoxes and TBoxes in SPARQL. *CoRR*, abs/1403.7248, 2014.

K. Cerans, G. Barzdins, R. Liepins, J. Ovcinnikova, S. Rikacovs, and A. Sprogis. Graphical schema editing for stardog OWL/RDF databases using OWLGrEd/S. 849, 01 2012.

J. Chabin, C. Eichler, M. Halfed Ferrari, and N. Hiot. SetUp: a tool for consistent updates of RDF knowledge graphs. https://www.univ-orleans.fr/lifo/evenements/sendup-project/index.php/softwares/setup-schema-evolution-through-updates/, 2019a.

J. Chabin, M. Halfeld Ferrari, and D. Laurent. Consistent updating of databases with marked nulls. *Knowledge and Information Systems*, 2019b.

J. Chabin, C. Eichler, M. Halfed Ferrari, and N. Hiot. Graph rewriting system for consistent evolution of RDF databases. Technical report, LIFO, Université d'Orléans, INSA Centre Val de Loire, 2020a. URL `hal.archives-ouvertes.fr/hal-02560325v3`.

J. Chabin, C. Eichler, M. Halfeld-Ferrari, and N. Hiot. Graph rewriting rules for rdf database evolution management. In *Proceedings of the 22nd International Conference on Information Integration and Web-Based Applications & Services*, iiWAS '20, page 134143, New York, NY, USA, 2020b. Association for Computing Machinery. ISBN 9781450389228. doi: 10.1145/3428757.3429126. URL `https://doi.org/10.1145/3428757.3429126`.

P. De Leenheer and T. Mens. Using graph transformation to support collaborative ontology evolution. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, pages 44–58, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-89020-1.

M. A. Farvardin, D. Colazzo, K. Belhajjame, and C. Sartiani. Scalable saturation of streaming RDF triples. *Trans. Large Scale Data Knowl. Centered Syst.*, 44:1–40, 2020.

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

F. Goasdoué, I. Manolescu, and A. Roatiş. Efficient query answering against dynamic RDF databases. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 299–310. ACM, 2013.

U. Golas, E. Biermann, H. Ehrig, and C. Ermel. A visual interpreter semantics for statecharts based on amalgamated graph transformation. *ECEASST*, 39, 01 2011.

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

A. Habel and K.-h. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical. Structures in Comp. Sci.*, 19(2):245–296, Apr. 2009. ISSN 0960-1295.

A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundam. Inf.*, 26(3,4):287–313, Dec. 1996.

M. Halfeld Ferrari and D. Laurent. Updating RDF/S databases under constraints. In *Advances in Databases and Information Systems - 21st European Conference, ADBIS, Nicosia, Cyprus, Proceedings*, pages 357–371, 2017.

M. Halfeld Ferrari, C. S. Hara, and F. R. Uber. RDF updates with constraints. In *Knowledge Engineering and Semantic Web - 8th International Conference, KESW, Szczecin, Poland, Proceedings*, pages 229–245, 2017.

S. O. Hansson. Logic of belief revision. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2016 edition, 2016.

H. Katsuno and A. O. Mendelzon. 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.*, pages 387–394, 1991.

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

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

S. Link. Towards a tailored theory of consistency enforcement in databases. In *Foundations of Information and Knowledge Systems, Second International Symposium, FoIKS, Germany, Proceedings*, pages 160–177, 2002.

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

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

M. Mahfoudh. *Adaptation d'ontologies avec les grammaires de graphes typés: évolution et fusion. (Ontologies adaptation with typed graph grammars : evolution and merging)*. PhD thesis, University of Upper Alsace, Mulhouse, France, 2015.

M. Mahfoudh, G. Forestier, L. Thiry, and M. Hassenforder. Algebraic graph transformations for formalizing ontology changes and evolving ontologies. *Knowledge-Based Systems*, 73:212 – 226, 2015.

P. Maillot, T. Raimbault, D. Genest, and S. Loiseau. 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*, pages 187–193, 2014.

O. Runge, C. Ermel, and G. Taentzer. Agg 2.0 – new features for specifying and analyzing algebraic graph transformations. In *Applications of Graph Transformations with Industrial Relevance*, pages 81–88, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

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

S. Segura, D. Benavides, A. Ruiz-Cortés, and P. Trinidad. *Automated Merging of Feature Models Using Graph Transformations*, pages 489–505. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

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

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

A. Shipilev, S. Kuksenko, A. Astrand, S. Friberg, and H. Loef. OpenJDK Code Tools: JMH, 2007. URL `https://openjdk.java.net/projects/code-tools/jmh/`.

E. Sirin, M. Smith, and E. Wallace. Opening, closing worlds - on integrity constraints. In C. Dolbear, A. Ruttenberg, and U. Sattler, editors, *Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions, collocated with the 7th International Semantic Web Conference (ISWC-2008)*, volume 432 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

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

G. Taentzer. Agg: A graph transformation environment for modeling and validation of software. In *AGTIVE*, 2003.

J. Tao, E. Sirin, J. Bao, and D. L. McGuinness. Integrity constraints in OWL. In M. Fox and D. Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, , USA*. AAAI Press, 2010.

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

M. Winslett. *Updating Logical Databases*. Cambridge University Press, New York, NY, USA, 1990.

A Proof of Lemma

COMMENT FOR REVIEWERS: The lemma is given to reviewer's appreciation. We put it in an appendix due to constraints on the paper size. It can be omitted in a published version.

Lemma 2 *Let $\mathbb{G}$ be a consistent graph and $\mathbb{R}$ our set of graph rewriting rules. Let $\mathscr{D}'/\mathbb{G}'$ be the database such that $\mathbb{G}' = \mathsf{SetUp}_{\text{OPT}}(\mathbb{G}, \mathbb{R}, U, U_{prev})$ where $U = PSub(A, B)$ and $U_{prev} = \emptyset$ are consistent updates. Then $\mathbb{G}'$ is a consistent graph where A is a sub-property of B.* □

PROOF. Update U is a schema update. When neither A nor B exist in the $\mathbb{G}$, we have a trivial situation: $\mathsf{SetUp}_{\text{OPT}}$ creates two properties setting A as sub-property of B. To consider situations where properties A or B exist in $\mathbb{G}$, let us define a hierarchy of properties as a set of properties $P_1, \ldots, P_n$ such that each P_i is a sub-property of P_{i+1} for $i \in [0, n-1]$.

Situation1: Both A and B exist and belong to the same hierarchy H (*i.e.*, there is a sequence of successive properties from A to B or from B to A). We consider two cases:
• (1.1) $A = P_i$ and $B = P_j$ for $i > j$. In this case, as $\mathbb{G}$ is consistent and complete, due to rule 20, there is a link $PSub(B, A)$ in $\mathbb{G}$ (*i.e.*, B is a sub-class of A). The insertion $U = PSub(A, B)$ is thus a contradiction detected by side effect $s5$ in Fig. 4(a). Side-effect $s5$ is implemented by reversing rule 20, as shown in Fig. 4(a). Different deletion choices can be applied here, but the consequence is always the same: after deletions, B is not a sub-property of A anymore. Priority is given to the update, thus A is set as a sub-property of B. In this case, we insert $PSub(A, B)$ in a graph where A belongs to an hierarchy, say H_1 and B to another one, say H_2 (the common hierarchy is broken by the deletions). This is the case we consider in Situation2 below.
• (1.2) $A = P_i$ and $B = P_j$ for $i < j$. In this case, following the same reasoning of (1.1) we know that there is a link $PSub(A, B)$ in $\mathbb{G}$ and thus the insertion $U = PSub(A, B)$ is superfluous.

Situation2: Both A and B exist such that $A \in H_1$ and $B \in H_2$ where H_1 and H_2 are two non-empty hierarchies in $\mathbb{G}$ (*i.e.*, H_1 and H_2 have at least the property being inserted). To position the properties in their hierarchy, let $A = P_{l_a}$ and $B = P_{l_b}$. Recall that side-effects $s6$ and $s7$ are implemented on the basis of rule 20 whose format is: $PSub(x, y) \wedge PSub(y, z) \Rightarrow PSub(x, z)$.
• (2.1) Under the instantiation $x \to A$ and $y \to B$, if there are super-properties z of B, then A is set as a sub-property of each super-property z of B.
• (2.2) Under the instantiation $y \to A$ and $z \to B$, if there are sub-properties x of A, then a link between x and B is created, *i.e.*, each sub-property x of A is set as a sub-property of B.
• (2.3) Now suppose $P_j \in H_2$ is one of the instantiations of z in (2.1). We have a new sub-property relationship $PSub(A, P_j)$ from (2.1). By using the same transitive rule 20, we should add $PSub(A, P_k)$ for all $k > j$ to our graph database. However, as $k > j \geq l_b$, we know that edges indicating this sub-property hierarchy have already been establish in (2.1). Thus, for generated updates of the form $PSub(A, P_j)$, no subsequent *schema* side-effect need to be considered.
• (2.4) Now suppose $P_j \in H_1$ ($j < l_a$) is one of the instantiations of x in (2.2). We have a new sub-property relationship $PSub(P_j, B)$ from (2.2). By using the same transitive rule 20, we add $PSub(P_j, P_k)$ (for all $j < l_a$ and $k > l_b$) to our graph database. The inclusion of these edges finishes the transitive closure obtained with rule 20.

(*) In conclusion, for performing the *schema* change required by the insertion $PSub(A, B)$, only step (2.4) above need to be considered when dealing with insertions, such as $PSub(P_j, B)$, which are side-effects of the original $PSub(A, B)$.

Impact on instances. On the other hand, steps (2.1) − (2.4) above generate sub-properties relationships which may trigger rule 27 concerning property instances. Indeed, a schema change may imply instance changes: the insertion of $PSub(A, B)$ also triggers side-effect $s8$ (Fig.4(a)). This side effect corresponds to the application of rule $PSub(z, w) \wedge PI(x, y, z) \Rightarrow PI(x, y, w)$. Firstly, for instantiation $z \to A$ and $w \to B$, all instances of property A are set as instances of property B. Then, similarly, for generated updates of type $PSub(A, P_j)$, $PSub(P_j, B)$ and $PSub(P_j, P_k)$ (steps (2.3) and (2.4) above), side-effect $s8$ is triggered to propagate instances of a

sub-property to its super-property. Note that the insertion of a property instance may also trigger rules 14, 24 and 25. However, when these insertions are triggered by the application of the rule $PSub(z, w) \wedge PI(x, y, z) \Rightarrow PI(x, y, w)$ the application of these constraints can be ignored. Indeed, typing (rule 14) has already been verified, since we have $PI(x, y, z)$ and $PSub(z, w)$ (i.e. x and y are individuals and w is a property). Denote by d_z, r_z, d_w, and r_w the domains and ranges of z and w, respectively. As we have $PSub(z, w)$, rules 21 and 23 ensure that $r_w = r_z$ (resp. $d_w = d_z$) or $CSub(r_z, r_w)$ (resp. $CSub(d_z, d_w)$. As we have $PI(x, y, z)$, rules 24 and 25 ensure $CI(x, d_z)$ and $CI(y, r_z)$. Therefore, if $CSub(r_z, r_w)$ (resp. $CSub(d_z, d_w)$, rule 26 imposes that $CI(y, r_w)$ (resp. $CI(x, d_w)$). If $r_w = r_z$ (resp. $d_w = d_z$), we have $CI(y, r_w)$ (resp. $CI(x, d_w)$).

(**) Thus, for performing the *instances* changes implied in the required insertion $PSub(A, B)$, side-effect $s8$ (rule 27) need to be considered – triggered by $PSub(A, P_j)$, $PSub(P_j, B)$ and $PSub(P_j, P_k)$, which are side-effects of the original $PSub(A, B)$. □