

Graph Rewriting Rules for RDF Database Evolution Management

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

▶ **To cite this version:**

Jacques Chabin, Cédric Eichler, Mirian Halfeld Ferrari, Nicolas Hiot. Graph Rewriting Rules for RDF Database Evolution Management. International Conference on Information Integration and Web-based Applications & Services, 2020, Chiang Mai (On line), Thailand. hal-02956728

HAL Id: hal-02956728
https://hal.science/hal-02956728

Submitted on 14 Oct 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Graph Rewriting Rules for RDF Database Evolution Management

Jacques Chabin
jchabin@univ-orleans.fr
Université d'Orléans, INSA CVL, LIFO EA
Orléans, France

Cédric Eichler
cedric.eichler@insa-cvl.fr
INSA CVL, Université d'Orléans, LIFO EA
Bourges, France

Mirian Halfeld-Ferrari
mirian@univ-orleans.fr
Université d'Orléans, INSA CVL, LIFO EA
Orléans, France

Nicolas Hiot
nicolas.hiot@univ-orleans.fr
Université d'Orléans, INSA CVL, LIFO EA
Orléans, France

ABSTRACT

This paper introduces SetUp, a theoretical and applied framework for the management of RDF/S database evolution on the basis of *graph rewriting rules*. Rewriting rules formalize instance or schema changes, ensuring graph's consistency with respect to given constraints. Constraints considered in this paper are a well known variant of RDF/S semantic, but the approach can be adapted to user-defined constraints. Furthermore, SetUp manages updates by ensuring rule applicability through the generation of side-effects: new updates which guarantee that rule application conditions hold. We provide herein formal validation and experimental evaluation of SetUp.

CCS CONCEPTS

• **Information systems → Graph-based database models**; • **Theory of computation → Rewrite systems**.

KEYWORDS

Graph rewriting, Database Management, Update, Constraints

ACM Reference Format:
Jacques Chabin, Cédric Eichler, Mirian Halfeld-Ferrari, and Nicolas Hiot. 2020. Graph Rewriting Rules for RDF Database Evolution Management. In *Proceedings of iiWAS '20: International Conference on Information Integration and Web-based Applications & Services (iiWAS '20).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXX

1 INTRODUCTION

Graph rewriting concerns the technique of transforming a graph. It is thus natural to conceive its application in the evolution of graph databases. This paper shows the utility of this formal tool into a practical and useful application, by proposing a framework which ensures the *consistent* evolution of RDF (Resource Description Framework) databases. Indeed, 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. "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 this paper, 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]. 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 1.1 (Motivating Example.). Fig. 1 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 paper 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. 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
iiWAS '20, November 30– December 02, 2020, Chiang Mai, Thailand
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXX

SetUp summarized in two main steps
(1) Firstly we formalize updates as *graph rewriting rules* encompassing integrity constraints. An *Update* is a general term and can be classified through two different aspects: on one hand, as insertions or deletions and, on the other hand as instance or schema changes. Each update is formalized by a graph rewriting rule whose application *necessarily* preserves the databases validity. To perform an update, the applicability conditions of the corresponding rule are automatically checked. When all conditions of a rule hold, the rule is activated to produce a new graph which takes into account the required update and is necessarily valid if the graph was valid prior to the update. 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 rules. When an evolution is mandatory, we enforce database evolution by performing *side-effects* (*i.e.*, triggering other updates or schema modifications which will render possible rule application).

SetUp's main characteristics.
• SetUp main goal is to ensure validity when dealing with the evolution of an RDF/S data graph which represents a set of RDF (the instance) and RDFS (the schema) triples respecting semantic constraints as defined in [9].
• SetUp deals with *complete* instances, *i.e.*, constraint satisfaction is obtained only when the required data is *effectively* stored in the database.
• SetUp implements deterministic rules. Arbitrary choices have been made when non-deterministic options are available.
• SetUp takes into account the user level. Only database administrators may require updates provoking schema changes.
Paper Organization. After some related work in Section 2, Section 3 sets up work context. Section 4 formalizes updates by rewriting rules, Section 5 deals with side-effects and Section 6 shows our experiments. Conclusions and perspectives are drawn in Section 7.

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 work can be considered as particularly original: (i) the use of graph rewriting techniques and (ii) the adoption of CWA with RDF data. 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.*, [7, 9, 15]).

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 propose [8, 25, 31]. They usually focus on formalization but do not provide an implementation. To the best of our knowledge, only [24] 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 [25]) with OWA under some consistency constraints. Nested and general application conditions are not considered in [24], 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).

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 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 apply to any graph databases where consistency has to be preserved. Thus, here, RDF is one possible graph data model. 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.

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 [17] 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 [7, 15], we consider updates as changes in the world rather than as a revision in our knowledge of the world ([17, 18]). 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, for instance, [21, 22]. In this paper, we only deal RDF/S constraints whose consistency is ensured; but our approach could be extended to deal with user-defined constraints.

Several recent works focus on consistent graph databases. The approach in [26] differs from ours, by proposing a semantic measure based on the difference between original and updated RDF sub-graph. Both [7, 10] consider RDF updating methods, but the former goes deeper in the study of null values. A parallel can be done between saturation in [10], the chase in [7, 9, 15] and *SetUp*. Authors in [7, 9, 10, 15] offer home-made procedures to implement their methods: [10] deals only with the RDF instance constraints (Fig. 2); in [7, 9], constraints are user's tuple-generating-dependencies. Incomplete information and updates are the focus of [7, 15]. Schema evolution is mentioned in [9, 10]. More expressive constraints represent a barrier to the update determinism: guarantee in [16] due

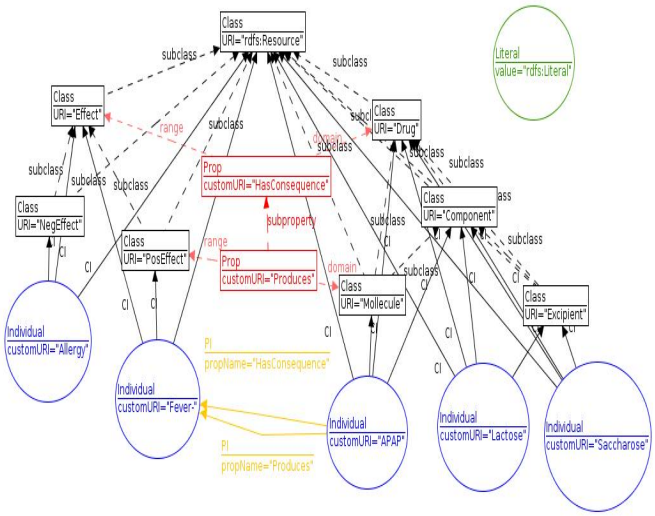


Figure 1: RDF schema and instance.

to simple rules and in [9] 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, 12] where constraints are just inference rules in OWA. Although some RDF technologies such as ShEx [34], SPIN [19], and SHACL [20] 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.

3 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.

In [9] 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 = $\{CI, PI, Ind\}$, 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.

Definition 3.1 (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. 1 shows an RDF instance and schema as a typed graph whose specifications are available in [6][1]. 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. Constraints presented in [9] are those in Fig. 2 which is borrowed from [16]. We recall from [9] 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 (20) establishes transitivity between sub-properties and the instance constraint (27) ensures this transitivity on instances of a property (if z is a sub-property of w, all z's instances are property instances of w). We are interested in database that satisfy all constraints in $\mathscr{C}$.

Definition 3.2 (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 3.3 (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. 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. 3: 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

[1] Roughly, nodes and edges are typed (represented with a unique combination of shape and color, *e.g.*, an individual node is a blue oval) and attributed to indicate a name or value when relevant.

• Typing Constraints:

$$CL(x) \Rightarrow URI(x) \quad (1) \qquad Pr(x) \Rightarrow URI(x) \quad (2) \qquad Ind(x) \Rightarrow URI(x) \quad (3)$$
$$(CL(x) \wedge Pr(x)) \Rightarrow \bot \quad (4) \qquad (CL(x) \wedge Ind(x)) \Rightarrow \bot \quad (5) \qquad (Pr(x) \wedge Ind(x)) \Rightarrow \bot \quad (6)$$
$$CSub(x, y) \Rightarrow CL(x) \wedge CL(y) \quad (7) \qquad PSub(x, y) \Rightarrow Pr(x) \wedge Pr(y) \quad (8) \qquad Dom(x, y) \Rightarrow Pr(x) \wedge CL(y) \quad (9)$$
$$Rng(x, y) \Rightarrow Pr(x) \wedge CL(y) \quad (10) \qquad CI(x, y) \Rightarrow Ind(x) \wedge CL(y) \quad (11) \qquad CL(x) \Rightarrow CSub(x, \textit{rdfs:Resource}) \quad (12)$$
$$Ind(x) \Rightarrow CI(x, \textit{rdfs:Resource}) \quad (13) \qquad PI(x, y, z) \Rightarrow Ind(x) \wedge Ind(y) \wedge Pr(z) \quad (14)$$

• Schema Constraints:

$$Pr(x) \Rightarrow (\exists y, z)(Dom(x, y) \wedge Rng(x, y)) \quad (15) \qquad ((y \neq z) \wedge Dom(x, y) \wedge Dom(x, z)) \Rightarrow \bot \quad (16) \qquad ((y \neq z) \wedge Rng(x, y) \wedge Rng(x, z)) \Rightarrow \bot \quad (17)$$
$$CSub(x, y) \wedge CSub(y, z) \Rightarrow CSub(x, z) \quad (18) \qquad CSub(x, y) \wedge CSub(y, x) \Rightarrow \bot \quad (19) \qquad PSub(x, y) \wedge PSub(y, z) \Rightarrow PSub(x, z) \quad (20)$$
$$Psub(x, y) \wedge Dom(x, z) \wedge Dom(y, w) \wedge (z \neq w) \Rightarrow CSub(z, w) \quad (21) \qquad PSub(x, y) \wedge PSub(y, x) \Rightarrow \bot \quad (22)$$
$$Psub(x, y) \wedge Rng(x, z) \wedge Rng(y, w) \wedge (z \neq w) \Rightarrow CSub(z, w) \quad (23)$$

• Instance Constraints:

$$Dom(z, w) \Rightarrow (PI(x, y, z) \Rightarrow CI(x, w)) \quad (24) \qquad Rng(z, w) \Rightarrow (PI(x, y, z) \Rightarrow CI(x, w)) \quad (25)$$
$$CSub(y, z) \Rightarrow (CI(x, y) \Rightarrow CI(x, z)) \quad (26) \qquad PSub(z, w) \Rightarrow (PI(x, y, z) \Rightarrow PI(x, y, w)) \quad (27)$$

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

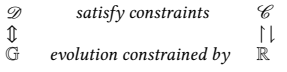


Figure 3: Rewriting rules $\mathbb{R}$ and constraints $\mathscr{C}$.

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 18 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 rule. All rules and proofs are available in [6]. The following presents the background on graph rewriting illustrated by a single rule of $\mathbb{R}$. We adopt the Single Push Out (SPO) formalism [23] to specify rewriting rule as well as several of its extensions to specify additional application conditions and restrict rule applicability: *Negative Application Conditions* (NACs) [13], *Positive Application Conditions* (PACs), and *General Application Conditions* (GACs) [27].

Example 4.1. Consider the graph database of Fig. 1 and assume node *Allergy* exists in $\mathbb{G}$ but is only connected to node *rdf:ressource* and not to nodes such as *Effect*. In this context, 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. 4). □

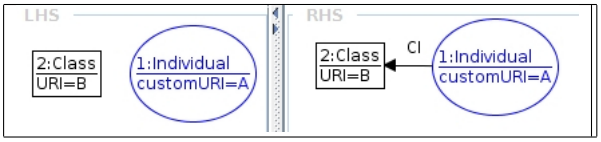


Figure 4: SPO rule: insertion of a class instance

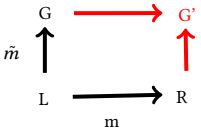


Figure 5: Push-Out, application of SPO rules

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 – and a partial morphism m from L to R (*i.e.*, an edge-preserving morphism m from an induced subgraph of L to R). [2]

Fig. 4 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 an 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. 4, 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 two morphisms $m : \mathsf{L} \rightarrow \mathsf{R}$ and $\tilde{m} : \mathsf{L} \rightarrow G$ are shown in black in the diagram of Fig. 5. 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 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

[2] 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.

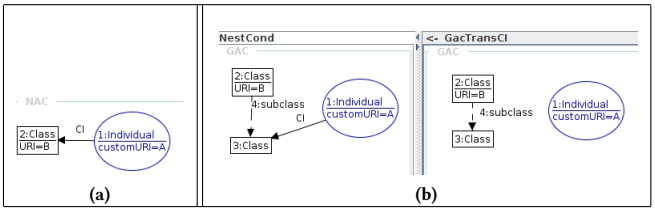


Figure 6: NAC and GAC for the SPO rule in Fig 4.

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 4.2. In Fig. 4, 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 a constraint graph which is a super-graph of L_r. An SPO rule $r = (L, R, m)$ with NACs and PACs is applicable to a graph iff: (i) there exists a total morphism $\tilde{m} : \mathsf{L} \rightarrow G$ (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) $\bar{m} : P \rightarrow G$ 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 4.3. Fig. 6a specifies a NAC for the rule in Fig. 4. 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* [11, 14] 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 $a : G \rightarrow C$ 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 (L, 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 [27] are a combination of nested application condition allowing the definition of complex application 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 $\bar{m} : G \rightarrow C$ such that $\bar{m} \circ a = \tilde{m}$ and $\bar{m}$ satisfies c.

Example 4.4. Fig. 6b 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. 6b. *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. 6b): *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 $CL(C)$). □

To prove that rule r_{CI}, defined by the SPO core of Fig. 4 and application conditions of Fig. 6, 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 (available in [6]). Based on that work we can also prove the following lemma.

Lemma 4.5 (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 $r_U \in \mathbb{R}$. If applicability conditions of r_U are not respected, our algorithm generates new updates capable of changing $\mathbb{G}$ into a new graph $\mathbb{G}^n$ on which r_U can be applied to produce $\mathbb{G}'$. These new updates are called side-effects of U.

Example 5.1. Let $\mathscr{D}/\mathbb{G}$ be the database as the one in Fig. 1, but without *NegEffect* and its incident edges. In this context, consider that U is the insertion of class instance $CI(Allergy, NegEffect)$ and $r_{CI} \in \mathbb{R}$ the corresponding rule (Fig. 4). 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". Thus, two *side-effects* are generated: (U^1) the insertion of an individual *Allergy* and (U^2) the insertion of class *NegEffect*. The corresponding rules are triggered, adding the individual and class and connecting

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

Figure 7: Extract of UpdCond table.

them to class *rdfs:Resource*. Rule r_{CI} is 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, 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. 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*.

Given a database $\mathscr{D}/\mathbb{G}$ and an update U, Algorithm 1 transforms $\mathbb{G}$ following rules in $\mathbb{R}$. Denote by $r_U \in \mathbb{R}$ the rewriting rule associated to U. When r_U cannot be applied on $\mathbb{G}$, SetUp computes, recursively, all updates necessary to change $\mathbb{G}$ so that r_U is applicable. Here, it is worth noting the design flexibility imposed by the update scenario: either imposed by the intrinsic non-determinism of consistency maintenance or by side-effect ordering.

Indeed, 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 [6] indexed by the update type. Fig. 7 shows an extract of UpdCond (*e.g.*, from the first row, we know that the insertion of $CI(A, B)$, depends on the existence of A as an individual, B as an class and the respect of hierarchical constraints). Roughly, to design UpdCond for an insertion P, we consider all constraints $c \in \mathscr{C}$ (Fig. 2) having atoms with the predicate of P in L_c (its body) and we build updates corresponding to the atoms in R_c (its head). Deletions are treated in a reciprocal way: we look from the predicate of P on the heads of constraints and define the new updates based on the atoms in their bodies. Unfortunately, a deletion may engender non-deterministic side-effects. Consider for instance the deletion of a class instance $CI(A, B)$. Constraint 26 in $\mathscr{C}$ (Fig. 2) indicates two possible side-effects in this case: deleting A as a class instance of all super-classes of B or breaking the class hierarchy. In this paper, we deal with non-determinism in an arbitrary way: when a choice is needed, the priority is given to updates on the instance, leaving the schema unchanged. Thus, in the above example, side-effect updates are: $\neg CI(A, y_c)$, for each y_c which is a super-class of B. When non-determinism is over two side-effects implying changes on the schema, the priority is to break the highest hierarchical link.

Then, on line 2 of Algorithm 1, each condition c is considered. The order in which each c is treated impacts the order in which new updates are applied to the database and gives rise to different approaches. *PreConditions* can be seen as a set (updates treated on any order) or as a list ordered according to a particular method. Our prototype uses an arbitrarily pre-defined order. Arbitrary choices are seldom the best solution: we are currently working on a cost function to guide our choices both for update ordering and non-deterministic choices.

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** $\mathbb{G}$

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 in the same way. When conditions for a rewriting rule $r_{u'}$ hold, function *GraphRewriter* applies $r_{u'}$ and the graph evolves. Eventually, if U is not intrinsically inconsistent, we obtain a new graph on which r_U is applicable. Indeed, some updates U related to a fact F may be intrinsically inconsistent as illustrated in the following example.

Example 5.2. Trying to perform SetUp $(\mathbb{G}, \mathbb{R}, CI(Sweetener, Sweetener))$, an inconsistent update, Algorithm 1, line 1, produces two pre-conditions $c1 : Ind(Sweetener)$ and $c2 : CL(Sweetener)$.
- $c1$ is not satisfied in $\mathbb{G}$, line 4 produces side-effect $Ind(Sweetener)$, which is inserted in the graph by line 6 with no more side-effect.
- $c2$ is not satisfied in $\mathbb{G}$ line 4 produces side-effect $CL(Sweetener)$, line 6 deletes $Ind(Sweetener)$ before applying the rule r_{CL} and insert $CL(Sweetener)$in $\mathbb{G}$.
Conditions c_1 and c_2 having been handled, r_{CI} is invoked but it cannot be applied: there is no individual node *Sweetener*; the algorithm stops. □

The current version of SetUp is simple and avoids loops in the treatment of intrinsically inconsistent updates. It performs updates 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. The resulting graph, in this case, does not change, but stays consistent.

Lemma 5.3 (SetUp Correction). *Let $\mathbb{G}$ be a consistent graph and $\mathbb{R}$ our set of graph rewriting rules. Let U be a consistent update, F the fact being inserted or deleted. Let $\mathscr{D}'/\mathbb{G}'$ be the database such that $\mathbb{G}' = SetUp(\mathbb{G}, \mathbb{R}, U)$. Then, (1) SetUp terminates and (2) if U is an insertion $F \in \mathscr{D}'$, otherwise, if U is a deletion $F \notin \mathscr{D}'$.* □

Proof. (sketch, full proof available in [6]) For each type of update we prove that we have a finite number of side-effects and thus recursive calls terminate. □

6 EXPERIMENTAL EVALUATION

SetUp is implemented in Java and relies on AGG (The Attributed Graph Grammar System) [35], one of the most mature and cited development environment supporting the definition and application of typed graph rewriting systems [29]. Its current version –available online [5]– provides a textual interface and offers different updating modes, according to the user level. This section experimentally investigates SetUp in various update scenarios, evaluating their execution time and the number of generated side-effects.

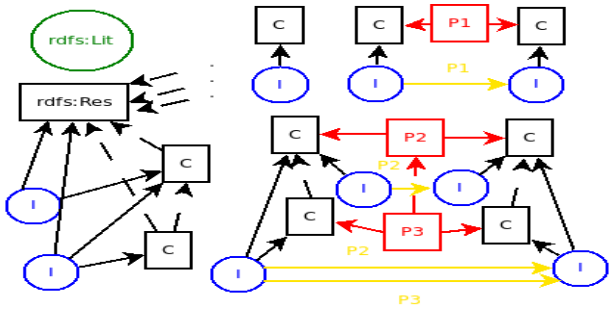


Figure 8: Experimental graph with $I = 1$ and $S = 2$

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 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. 8 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. 8) 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 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 $\exists P_{=CinH}$).

Experimental results. Figs. 9 and 10 show the results of our experiments for deletions and insertions, respectively. The time is measured with JMH [32] on 3 forks of 10 warmups and 50 measure iterations with a mean of 150 op./iteration.

Side-effects tackled by the *GraphRewriter* are not taken into account: for instance, the deletion of a class at the top of the hierarchy is reported with 0 side-effect since deletion of relevant CI and $CSub$ are handled by the *GraphRewriter* through the removal of dangling edges. On the contrary, those generated by SetUp are counted even if they do not need to be applied due to the database configuration. For instance the insertion $CL(A)$ has two side-effects ($\neg Pr(A)$ and $\neg Ind(A)$) that are included even if the original database does contain neither such a property nor such an individual. The number of generated side-effects varies according to the update type and the scenario. For instance, $CL(A)$ always generates the 2 aforementioned side-effects. As $CSub$ and CI relationships are suppressed by the *GraphRewriter*, update $\neg CL(C)$ generates 0 side-effects in scenarios $\neg\exists C$, $\exists C$, and $\exists C_{inH}$. However, in the scenario $\neg Dom/Rng$ top, 2, 46, 6, and 226 are generated with $(I, S) = (1, 1)$, $(1, 5)$, $(5, 1)$, and $(5, 5)$, respectively. The first generated side-effect is $\neg Pr(P)$ with P the property whose domain or range is suppressed. It itself generates $S * I$ $\neg PI$ (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.

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, 5s and 34, 4s 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, 5s) to $I = 5$ (660, 5s). By roughly tripling the size of the graph, each update $\neg CI$ takes almost 72% more time to be executed.

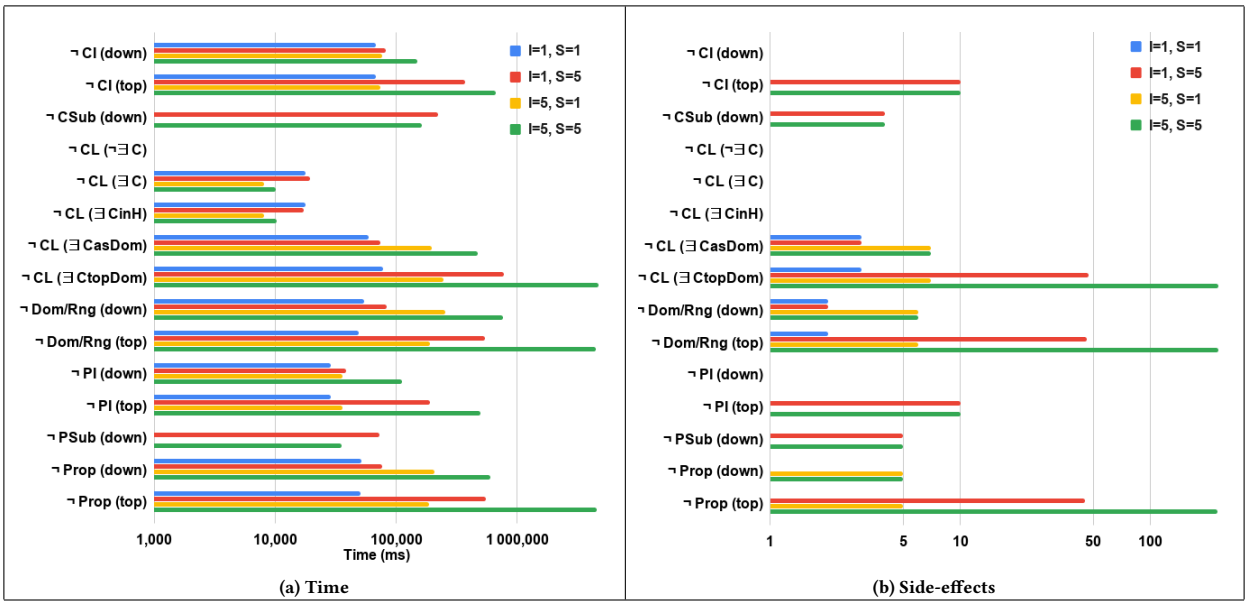
The second factor impacting time is the number of generated side-effects, as they triggers calls to the pattern matching and graph rewriting algorithms. For instance, configuration $(I, S) = (1, 5)$ is bigger than $(5, 1)$ as it as almost as many nodes but twice as many edges. Yet, $\neg Dom/Rng$ (top) is almost three times longer on the second configuration (190, 5s 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, 3s respectively with $I = S = 5$), even tough the latter implies the suppression of S $CSub$ relationships.

7 CONCLUSIONS AND PERSPECTIVES

SetUp is a theoretical and applied framework for ensuring consistent evolution of RDF graphs. Its 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 may

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

Scenario		*Update type*												
Notation	*Explanation*	$\neg CI$	$\neg CL$	$\neg CSub$	$\neg PI$	$\neg Prop$	$\neg PSub$	$\neg Dom$	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								✓					

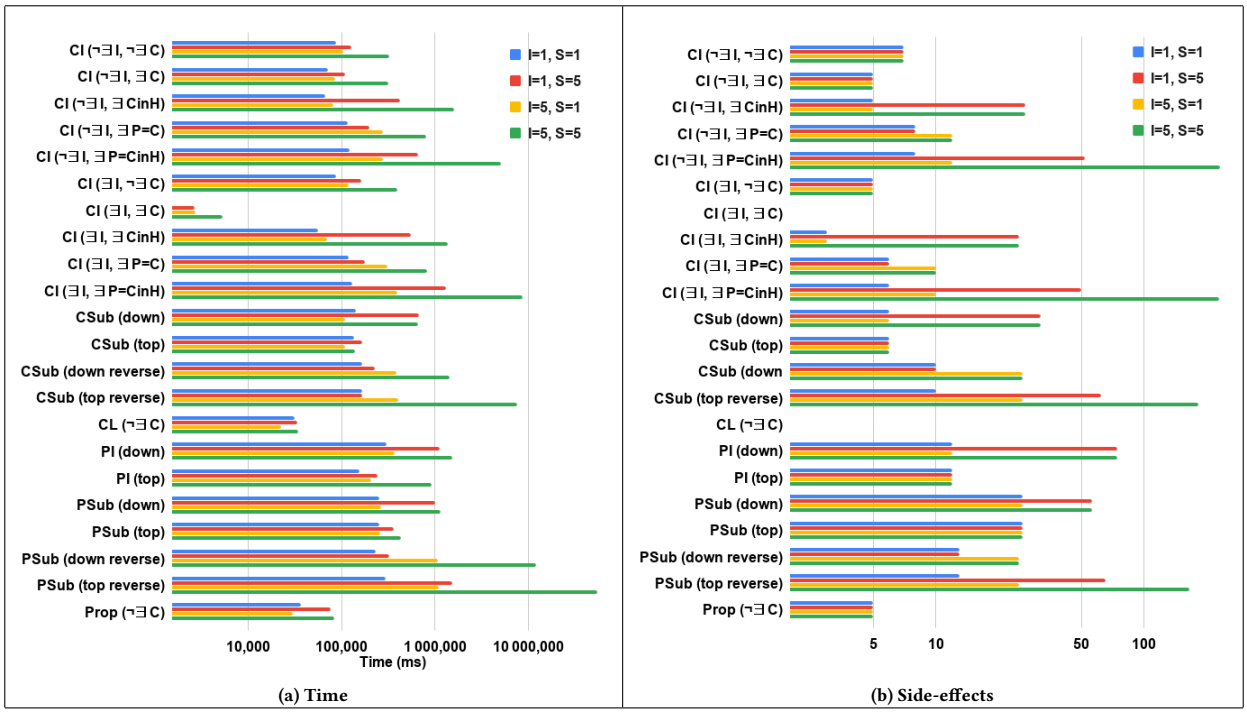


(a) Time (b) Side-effects

Figure 9: Experimental results for fact suppression.

generate additional consistency preserving updates to ensure its applicability.

The importance of SetUp is in its originality of using graph rewriting techniques under the closed world assumption to set an updating system. Even if it is unfit for on-the-fly automated updates, SetUp 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, we plan to use it 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 SetUp to conform to a privacy model such as k-anonymity or differential privacy. The advantage of SetUp to conduct such operations is threefold. First, even though the produced



(a) Time (b) Side-effects

Figure 10: Experimental results for fact insertion.

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. Secondly, 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. Thirdly, 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.

Acknowledgement: Work supported by the French National Research Agency, under grant ANR-18-CE23-0010 and developed in the context of DOING@DIAMS group.

REFERENCES

[1] [n.d.]. https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf/.
[2] Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, and Pierre Senellart. 2011. *Web data management.* Cambridge University Press.
[3] Albin Ahmeti, Diego Calvanese, and Axel Polleres. 2014. Updating RDFS ABoxes and TBoxes in SPARQL. *CoRR* abs/1403.7248 (2014).
[4] Karlis Cerans, Guntis Barzdins, Renars Liepins, Julija Ovcinnikova, Sergejs Rikacovs, and Arturs Sprogis. 2012. Graphical schema editing for StarDog OWL/RDF databases using OWLGrEd/S. 849 (01 2012).
[5] Jacques Chabin, Cédric Eichler, Mirian Halfed Ferrari, and Nicolas Hiot. 2019. SetUp: a tool for consistent updates of RDF knowledge graphs. https://owncloud.insa-cvl.fr/s/4w8eiqO2xDPBDiS.
[6] Jacques Chabin, Cédric Eichler, Mirian Halfed Ferrari, and Nicolas Hiot. 2020. *Graph rewriting system for consistent evolution of RDF databases.* Technical Report. LIFO, Université d'Orléans, INSA Centre Val de Loire. hal.archives-ouvertes.fr/hal-02560325v3
[7] Jacques Chabin, Mirian Halfeld Ferrari, and Dominique Laurent. 2019. Consistent Updating of Databases with Marked Nulls. *Knowledge and Information Systems* (2019).
[8] Pieter De Leenheer and Tom Mens. 2008. Using Graph Transformation to Support Collaborative Ontology Evolution. In *Applications of Graph Transformations with Industrial Relevance*, Andy Schürr, Manfred Nagl, and Albert Zündorf (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–58.
[9] Giorgos Flouris, George Konstantinidis, Grigoris Antoniou, and Vassilis Christophides. 2013. Formal foundations for RDF/S KB evolution. *Knowl. Inf. Syst.* 35, 1 (2013), 153–191.
[10] François Goasdoué, Ioana Manolescu, and Alexandra Roatiş. 2013. Efficient query answering against dynamic RDF databases. In *Proceedings of the 16th International Conference on Extending Database Technology.* ACM, 299–310.
[11] Ulrike Golas, Enrico Biermann, Hartmut Ehrig, and Claudia Ermel. 2011. A Visual Interpreter Semantics for Statecharts Based on Amalgamated Graph Transformation. *ECEASST* 39 (01 2011).
[12] Claudio Gutierrez, Carlos A. Hurtado, and Alejandro A. Vaisman. 2011. RDFS Update: From Theory to Practice. In *The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC, Greece, Proceedings, Part II.* 93–107.
[13] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. 1996. GRAPH GRAMMARS WITH NEGATIVE APPLICATION CONDITIONS. *Fundam. Inf.* 26, 3,4 (Dec. 1996), 287–313.
[14] Annegret Habel and Karl-heinz Pennemann. 2009. Correctness of High-level Transformation Systems Relative to Nested Conditions. *Mathematical. Structures in Comp. Sci.* 19, 2 (April 2009), 245–296.
[15] Mírian Halfeld Ferrari, Carmem S. Hara, and Flavio R. Uber. 2017. RDF Updates with Constraints. In *Knowledge Engineering and Semantic Web - 8th International*

Conference, KESW, Szczecin, Poland, Proceedings. 229–245.

[16] Mirian Halfeld Ferrari and Dominique Laurent. 2017. Updating RDF/S Databases Under Constraints. In *Advances in Databases and Information Systems - 21st European Conference, ADBIS, Nicosia, Cyprus, Proceedings*. 357–371.

[17] Sven Ove Hansson. 2016. Logic of Belief Revision. In *The Stanford Encyclopedia of Philosophy* (winter 2016 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.

[18] Hirofumi Katsuno and Alberto O. Mendelzon. 1991. 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*. 387–394.

[19] H. Knublauch, J. A. Hendler, and K. Idehen. 2011. SPIN - Overview and Motivation. W3C Member Submission. http://www.w3.org/Submission/2011/SUBM-spin-overview-20110222.

[20] H. Knublauch and A. Ryman. 2017. Shapes Constraint Language (SHACL). W3C First Public Working Draft, W3C. http://www.w3.org/TR/2015/WD-shacl-20151008/..

[21] Sebastian Link. 2002. Towards a Tailored Theory of Consistency Enforcement in Databases. In *Foundations of Information and Knowledge Systems, Second International Symposium, FoIKS, Germany, Proceedings*. 160–177.

[22] Sebastian Link and Klaus-Dieter Schewe. 2002. An Arithmetic Theory of Consistency Enforcement. *Acta Cybern.* 15, 3 (2002), 379–416.

[23] Michael Löe. 1993. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science* 109, 1âĂŞ2 (1993), 181 – 224.

[24] Mariem Mahfoudh. 2015. *Adaptation d'ontologies avec les grammaires de graphes typés: évolution et fusion. (Ontologies adaptation with typed graph grammars : evolution and merging)*. Ph.D. Dissertation. University of Upper Alsace, Mulhouse, France.

[25] Mariem Mahfoudh, Germain Forestier, Laurent Thiry, and Michel Hassenforder. 2015. Algebraic graph transformations for formalizing ontology changes and evolving ontologies. *Knowledge-Based Systems* 73 (2015), 212 – 226.

[26] Pierre Maillot, Thomas Raimbault, David Genest, and Stéphane Loiseau. 2014. 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*. 187–193.

[27] Olga Runge, Claudia Ermel, and Gabriele Taentzer. 2012. AGG 2.0 – New Features for Specifying and Analyzing Algebraic Graph Transformations. In *Applications of Graph Transformations with Industrial Relevance*. Springer Berlin Heidelberg, Berlin, Heidelberg, 81–88.

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

[29] Sergio Segura, David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. 2008. *Automated Merging of Feature Models Using Graph Transformations*. Springer Berlin Heidelberg, Berlin, Heidelberg, 489–505.

[30] Giorgos Serfiotis, Ioanna Koffina, Vassilis Christophides, and Val Tannen. 2005. Containment and Minimization of RDF/S Query Patterns. In *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)*, Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen (Eds.). Springer, 607–623.

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

[32] Aleksey Shipilev, Sergey Kuksenko, Anders Astrand, Staffan Friberg, and Henrik Loef. 2007. OpenJDK Code Tools: JMH. https://openjdk.java.net/projects/code-tools/jmh/

[33] Evren Sirin, Michael Smith, and Evan Wallace. 2008. Opening, Closing Worlds - On Integrity Constraints. In *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)*, Catherine Dolbear, Alan Ruttenberg, and Ulrike Sattler (Eds.). CEUR-WS.org.

[34] H. Solbrig and E. Prud'hommeaux. 2014. Shape Expressions 1.0 Definition. W3C Member Submission. http://www.w3.org/Submission/2014/SUBM-shex-defn-20140602.

[35] Gabriele Taentzer. 2003. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *AGTIVE*.

[36] Jiao Tao, Evren Sirin, Jie Bao, and Deborah L. McGuinness. 2010. Integrity Constraints in OWL. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, , USA*, Maria Fox and David Poole (Eds.). AAAI Press.

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

[38] Marianne Winslett. 1990. *Updating Logical Databases*. Cambridge University Press, New York, NY, USA.