Semantic Web -1 (2024) 1–41
DOI 10.3233/SW-243675
IOS Press
CORRECTED PROOF

PRSC: From PG to RDF and back, using schemas

Julian Bruyat [a,*], Pierre-Antoine Champin [a,b,c], Lionel Médini [a] and Frédérique Laforest [a]
[a] *INSA Lyon, CNRS, Universite Claude Bernard Lyon 1, LIRIS, UMR5205, 69621 Villeurbanne, France*
E-mails: julian.bruyat@liris.cnrs.fr, lionel.medini@liris.cnrs.fr, frederique.laforest@liris.cnrs.fr
[b] *W3C, Sophia Antipolis, France*
E-mail: pierre-antoine@w3.org
[c] *University Côte d'Azur, Inria, CNRS, I3S UMR 7271, France*

Editor: Stefan Schlobach, Vrije Universiteit Amsterdam, Netherlands
Solicited reviews: Olaf Hartig, Linköping University, Sweden; anonymous reviewer; anonymous reviewer

Abstract. Property graphs (PG) and RDF graphs are two popular database graph models, but they are not interoperable: data modeled in PG cannot be directly integrated with other data modeled in RDF. This lack of interoperability also impedes the use of the tools of one model when data are modeled in the other.

In this paper, we propose PRSC, a configurable conversion to transform a PG into an RDF graph. This conversion relies on PG schemas and user-defined mappings called PRSC contexts. We also formally prove that a subset of PRSC contexts, called well-behaved contexts, can be used to reverse back to the original PG, and provide the related algorithm. Algorithms for conversion and reversion are available as open-source implementations.

Keywords: Property Graph, RDF graph, conversion, schema

1. Introduction

Graphs are a popular data model in which knowledge is represented through objects and links between these objects. Today, there are two mainly used models of graphs: Property Graphs and RDF.

Property Graphs (PGs) are a family of implementations, in which data are represented with nodes and edges, and labels and properties (key-value pairs) can be attached to these nodes and edges. Property Graphs are not a uniform model: some implementations like Neo4j [1] only allow exactly one label for each edge. Most PG engines offer easy-to-use graph query languages like Cypher [16] and Gremlin [28] that rely on graph traversal. While no uniform standard has been settled yet, the *Property Graph needs a Schema Working Group* [2] is working towards defining a schema language for PG, and a unified formalization of the PG model. In the rest of this paper, following other authors [2], the variability of implementations is neglected and PGs are considered as a uniform model.

Another popular graph model is the Resource Description Framework (RDF) model [34]. In this model, data are represented with triples that represent links between resources. The resources and the links between them are

*Corresponding author. E-mail: julian.bruyat@liris.cnrs.fr.
[1] https://neo4j.com/
[2] https://www.w3.org/Data/events/data-ws-2019/assets/position/Juan%20Sequeda.txt

identified through Internationalized Resource Identifiers (IRI). This data model is a W3C standard, and has been studied through a large quantity of works, like RDFS [9] and OWL [24] for inference, or SHACL [22] for data validation. This model has been extended by RDF-star [20] that helps writing properties on triple terms in a more concise manner, but does not provide exactly the same modeling capabilities as PGs. For example, multiple links with the same predicate between two resources are still not supported and requires using extra constructs, like the occurrence pattern described in the "Triples and occurrences" section of the RDF-star community group final report [19].

While PG and RDF are both based on the idea of using graph data, the choice of one removes the ability to use the tools developed for the other. The maintainers of Amazon Neptune, a graph database service that can support both models independently, report that their users choose the solution that best suits their current use case, then struggle because they are stuck with the tools of this model even if the tools of the other model would better answer the new business problem they have [23]. Generally speaking, this diversity of graph models, and more precisely the lack of interoperability, hinders graph database adoption.

Numerous authors [14,29] highlight the benefits of interoperable systems, and in particular of interoperability between PGs and RDF graphs [15,32,35]. We can distinguish two levels of interoperability: syntactic interoperability and semantic interoperability. The former consists in having common data formats, for example JSON or CSV. The latter consists in having data not only in a common data format, but also that use shared vocabularies. Tim Berners-Lee theorized the 5-star Open-data model.[3] In this model, datasets are given a number of stars depending on how interoperable they are. RDF graphs are by design the best candidate to elevate the ranking of data in the 5-star data-model, in particular through the use of shared vocabularies. So, while internally developers may choose RDF or PG databases based on their preferences and the convenience of associated tools, RDF is better suited as a pivot model for interoperable exchange of data.

Among the many questions risen by the interoperability between PGs and RDF graphs, the scenario we want to support is the conversion from PGs to RDF (syntactic interoperability) without information loss, and while promoting the use of existing shared vocabularies (semantic interoperability). The requirement of using of existing shared vocabularies requires a converter to be highly customizable, in particular in terms of how the produced triples are structured. On the other hand, the requirement of not losing information, *i.e.* producing reversible conversions, requires to formally study the possible conversions.

In a previous paper [10], we introduced the motivations behind PREC (PG to RDF Experimental Converter), a user-configured mapping from Property Graphs (PG) to RDF graphs and proposed a mapping language, specialized for the PG to RDF conversion problem, to let the user describe how to convert the node labels, the edges and the properties of the original PG to an RDF graph. By converting the data stored in PGs into RDF, users are then able to use all the tools available for RDF. In this paper, we introduce a new mapping language, named PRSC,[4] driven by a schema and a description of how to convert the elements of the types in the schema to RDF. This mapping language is formally defined, and conditions under which the conversion produced by PRSC is reversible are also defined. Compared to the previous mapping language, this language contains fewer terms and concepts, and should be easier to use. The PRSC engine is available under the MIT license[5] and can connect both to a Cypher endpoint and a Gremlin endpoint.

The rest of this paper is organized as follows. Section 2 gives an overview of PRSC to understand its principles. Section 3 gives generic formal definitions of PGs and RDF graphs. Section 4 gives a formal definition of a PRSC conversion, which is essentially a formal definition of Section 2. Section 5 studies reversibility, and proves that some contexts can convert PG to RDF graph without information loss. Section 6 discusses the existing works to make easier interoperability between PGs and RDF graphs relatively to PRSC. Section 7 discusses the proposed solution and describes some future works.

[3] https://5stardata.info/en/

[4] PG to RDF: Schema-driven Converter, pronounced "presque".

[5] https://github.com/BruJu/PREC, https://npmjs.com/package/prec.

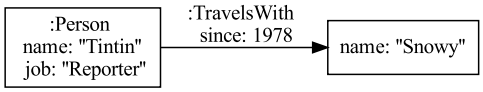

Fig. 1. A small PG about Tintin that serves as a running example in this paper.

```
1       _:n1 rdf:type ex:Person .
2       _:n1 foaf:name "Tintin" .
3       _:n1 ex:profession "Reporter" .
4       _:n1 ex:isTeammateOf _:n2 .
5       << _:n1 ex:isTeammateOf _:n2 >> ex:since 1978 .
6       _:n2 foaf:name "Snowy" .
```

Listing 1. An example of an RDF-star graph in Turtle Format

2. PRSC in practice

The Property Graph exposed on Fig. 1 describes the relationship between Tintin and Snowy. It is composed of two nodes. The first one holds the label Person and two properties: the first property has “name” as its key and “Tintin” as its value, the second has “job” as its key and “Reporter” as its value, or more simply its name is “Tintin” and its job is “Reporter”. The other node only has one property: the name “Snowy”. These two nodes are connected by an edge that holds one label, TravelsWith, and a property that tells that it is “since” “1978”.

A similar example represented in RDF-star is exposed on Listing 1. Most information that was in the PG is represented by the triples in lines 1–4 and 6. The information about since when Tintin travels with Snowy is represented through a nested RDF-star triple.

Using the user-defined mapping, PRSC is able to convert the PG in Fig. 1 into the RDF-star graph in Listing 1, and more generally any Property Graph with the same schema into the corresponding RDF graph. In this paper, we consider that Property Graphs with the same schema as the one in Fig. 1 are Property Graphs where all nodes have either (1) the “Person” label, a “name” property and a “job” property, (2) no label and a “name” property, (3) and where all edges have the “TravelWith” label and a “since” property. The mapping the user must provide to the PRSC engine is exposed on Listing 2 and is in Turtle-star format [27]. Rules are split in two parts:

- The target part that describes which elements of the Property Graph are targeted. The target is described depending on three criteria: (1) whether the element must be an edge or a node, (2) the labels and (3) the properties of the element.
- The production part that describes the triples to produce with a list of *template triples*. Values in the *pvar* namespace are mapped to the blank node in the resulting RDF graph. The literals that use *valueOf* as their datatype are converted to the property values in the RDF graph.

The mapping, named *PRSC context*, exposed on Listing 2 reads as follows:

- The first rule is named `_:PersonRule` (line 1)
  - The rule is used for all PG nodes (line 3) that only have the node label “Person” (line 4) and have the properties “name” and “job” (line 5). In our example, the node corresponding to Tintin matches this description, but Snowy does not as it misses the Person label and the job property.
  - It will produce three triples:
    - One triple with a blank node as its subject, `rdf:type` as its predicate and `ex:Person` as its object (line 8). Each node from the Property Graph is identified by a distinct blank node. In this example, `_:n1 rdf:type ex:Person` will be produced.
    - Another triple with the same blank node as its subject, `foaf:name` as its predicate and a literal that matches the value of the name property in the PG (line 9). The PRSC engine converts all literals whose datatype is `prec:valueOf` into the value of the corresponding property in the PG. In this example, `_:n1 rdf:type "Tintin"` will be produced.

```
1  _:PersonRule
2    # Target: all nodes with label "Person" and two properties "name" and "job"
3        a prec:PRSCNodeRule ;
4        prec:label "Person" ;
5        prec:propertyKey "name", "job" ;
6    # Production part of the rule: a template graph
7        prec:produces
8          << pvar:self rdf:type      ex:Person >> ,
9          << pvar:self foaf:name     "name"^^prec:valueOf >> ,
10         << pvar:self ex:profession "job"^^prec:valueOf >> .
11
12 _:NamedRule
13   # Target: all nodes with no label and one property "name"
14       a prec:PRSCNodeRule ;
15       prec:propertyKey "name" ;
16   # Production part of the rule
17       prec:produces
18         << pvar:self foaf:name "name"^^prec:valueOf >> .
19
20 _:TravelsWithRule
21   # Target: all edges with the label "TravelsWith" and one property "since"
22       a prec:PRSCEdgeRule ;
23       prec:label "TravelsWith" ;
24       prec:propertyKey "since" ;
25   # Production part of the rule
26       prec:produces
27           << pvar:source ex:isTeammateOf pvar:destination >>  ;
28         << << pvar:source ex:isTeammateOf pvar:destination >> ex:since "since"^^prec:valueOf >>.
```

Listing 2. The PRSC context that maps the PG running example to the RDF graph running example

  - ⋆ One last triple is produced with the same blank node as its subject, `ex:profession` as its predicate and a literal corresponding to the value of the property job (line 10). In this example, `_:n1 ex:profession "Reporter"` will be produced.

- The second rule is named `_:NamedRule` (line 12).
  - ∗ It is applied to nodes (line 14) that have no labels and only one property: name (line 15). This is the case of the PG node used to describe Snowy but not the one that describes Tintin as it has an extra label and an extra property.
  - ∗ These PG nodes will be converted into one triple with a blank node that identifies the PG node as its subject, `foaf:name` as its predicate and the literal that correspond to the value of the name property as its object (line 18). In this example, the triple `_:n2 foaf:name "Snowy"` is produced.
- The third rule is named `_:TravelsWithRule` (line 20):
  - ∗ It is used to convert edges (line 22) whose only label is “TravelsWith” (line 23) and with one and only one property named “since” (line 24).
  - ∗ These edges are converted by producing a triple with the identifier of the source PG node as the subject, `ex:isTeammateOf` as the predicate and the identifier of the destination PG node as the object (line 27). In this example, the triple `_:n1 ex:isTeammateOf _:n2` is produced.
  - ∗ A triple with a quoted triple is created by the rule on line 28: the triple that was created by the line 27 is used on the subject position of the triples created by this triple, `ex:since` is used as the predicate and the value of the “since” property is used as the object. In our example, the triple `<< _:n1 ex:isTeammateOf _:n2 >> ex:since 1978` is produced.
  - ∗ Note that in this example, `pvar:self` is not used in lines 27 and 28. If it was used, it would be mapped to a blank node that identifies the edge. The consequence of not using it is a smaller RDF graph, at the cost of if several PG edges with the “TravelsWith” label were present between the two same nodes, the RDF representation of these edges would have been merged.

Note that this mechanism of using quoted triples to describe templates in a pure Turtle-star file was already presented in our previous work [10]. Compared to R2RML [14], it lets the user describe the triples to produce with

a syntax closer to the triples that will actually be produced, but it is currently impossible to express templated terms in any position. For example, if the user wants to generate a named node <http://example.org/person/{name}> in subject position in which the {name} part is substituted with the value of the name property, R2RML offers it as a native feature. As PRSC is currently unable to generate arbitrary IRIs, nodes and edges from the PG are always mapped to blank nodes and minting IRIs is left for future works.

3. General definitions

This section introduces some standard definitions, mostly inspired from previous works.

3.1. Notations and conventions

Let *Str* be the set of all strings. Strings are noted between quotes. For example, "*node*", "*edge*", "*since*" and "*Snowy*" are strings.

Definition 1 (Domain and image of a function)**.** For all partial functions $f : D \to A$, *Dom* and *Img* are defined as follows:

- $Dom(f) = \{x | \exists y \in A, \text{ such that } f(x) = y\}$
- $Img(f) = \{y | \exists x \in D, \text{ such that } f(x) = y\}$.

Example 1. For the partial function $inverse : \mathbb{R} \to \mathbb{R}$, with $inverse(x) = 1/x$, $Dom(inverse) = Img(inverse) = \mathbb{R} - \{0\}$.

Let S be a set, we recall that 2^S denotes the set of all parts of S.

3.2. Compatible functions

For all functions f, we recall that they can be seen as sets: $f = \{(x, f(x)) | x \in Dom(f)\}$. For all sets S of 2-tuples, S can be seen as a function iff (if and only if) $\forall (x, y_1, y_2), (x, y_1) \in S \wedge (x, y_2) \in S \Rightarrow y_1 = y_2$.

Example 2. Consider the three functions f_1, f_2, f_3 exposed in Table 1.

As f_1, f_2 and f_3 can be defined with a set, it is possible to use the usual set operations.

The set $f_1 \cup f_2 = \{(-2, 66), (-1, 33), (0, 0), (1, 1)\}$ is a function: the first element of all tuples has a different value. Using a function notation, it may be written as:

$$(f_1 \cup f_2)(x) = \begin{cases} 66 & \text{if } x = -2 \quad [f_2(-2) = 66] \\ 33 & \text{if } x = -1 \quad [f_2(-1) = 33] \\ 0 & \text{if } x = 0 \quad [f_1(0) = f_2(0) = 0] \\ 1 & \text{if } x = 1 \quad [f_1(1) = 1] \end{cases}$$

Table 1

Some functions defined both with the usual function notation and with a set notation

Function notation	Set notation
$f_1(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } x = 1 \end{cases}$	$f_1 = \{(0, 0), (1, 1)\}$
$f_2(x) = \begin{cases} 66 & \text{if } x = -2 \\ 33 & \text{if } x = -1 \\ 0 & \text{if } x = 0 \end{cases}$	$f_2 = \{(-2, 66), (-1, 33), (0, 0)\}$
$f_3(x) = \begin{cases} 10 & \text{if } x = 0 \\ 1 & \text{if } x = 1 \end{cases}$	$f_3 = \{(0, 10), (1, 1)\}$

On the opposite, $f_1 \cup f_3 = \{(0, 0), (0, 10), (1, 1)\}$ is not a function. Both $(0, 0)$ and $(0, 10)$ are members of the set $f_1 \cup f_3$, $(f_1 \cup f_3)(0)$ would be equal to both $f_1(0) = 0$ and $f_3(0) = 10$ which are different values.

Remark 1. Instead of using the notation $\{(x_0, f(x_0)), (x_1, f(x_1)), \dots\}$, the notation $\{x_0 \mapsto f(x_0), x_1 \mapsto f(x_1), \dots\}$ is sometimes used to clarify the fact that a set is a function. For example, f_3 may be noted as $f_3 = \{0 \mapsto 10, 1 \mapsto 1\}$.

Definition 2 (Functions compatibility)**.** Two functions f and g are compatible iff $f \cup g$ is a function, *i.e.* $\forall (x, y_f, y_g), (x, y_f) \in f \wedge (x, y_g) \in g \Rightarrow y_f = y_g$.

In other words, two functions f and g are compatible iff for every common input, they share the same output *i.e.* $\forall x \in Dom(f) \cap Dom(g),\ f(x) = g(x)$.

3.3. Property Graph

Definition 3 (Property Graph)**.** Following the definition of Angles [2], a property graph pg is defined as the tuple $(N_{pg}, E_{pg}, src_{pg}, dest_{pg}, labels_{pg}, properties_{pg})$, where:

- N_{pg} and E_{pg} are finite sets with $N_{pg} \cap E_{pg} = \emptyset$. N_{pg} and E_{pg} are respectively the set of nodes and the set of edges of the property graph pg.
- $src_{pg} : E_{pg} \to N_{pg}$ and $dest_{pg} : E_{pg} \to N_{pg}$ are two total functions. These two functions map each edge to its starting and destination nodes.
- $labels_{pg} : N_{pg} \cup E_{pg} \to 2^{Str}$ is a total function. This function maps the nodes and edges to their sets of labels.
- $properties_{pg} : (N_{pg} \cup E_{pg}) \times Str \to V$ is a partial function. This function describes the properties of the elements. V is the set of all possible property values. Considering that a property is a key-value pair, it expects two inputs: a node or an edge, and a property key. The output is the property value.

The set of all property graphs is denoted PGs.

In this paper, property graph nodes and edges are grouped under the term **element**.

In the rest of the paper, when any PG x is introduced, we allow ourselves to use the symbols N_x, E_x, src_x, $dest_x$, $labels_x$ and $properties_x$ without explicitly defining x as the tuple $(N_x, E_x, src_x, dest_x, labels_x, properties_x)$.

Example 3 (Running example of a Property Graph)**.** The PG exposed on Fig. 1 can formally be defined as the PG denoted TT with

- $N_{TT} = \{n_1, n_2\}$; $E_{TT} = \{e_1\}$
- $src_{TT} = \{e_1 \mapsto n_1\}$; $dest_{TT} = \{e_1 \mapsto n_2\}$
- $labels_{TT} = \{n_1 \mapsto \{\text{“}Person\text{”}\}; n_2 \mapsto \emptyset; e_1 \mapsto \{\text{“}TravelsWith\text{”}\}\}$
-

$$properties_{TT} = \left\{ \begin{array}{c} (n_1, \text{“}name\text{”}) \mapsto \text{“}Tintin\text{”}; (n_1, \text{“}job\text{”}) \mapsto \text{“}Reporter\text{”} \\ (n_2, \text{“}name\text{”}) \mapsto \text{“}Snowy\text{”}; (e_1, \text{“}since\text{”}) \mapsto 1978 \end{array} \right\}$$

Definition 4 (The empty PG)**.** The empty PG, which is the PG that contains no nodes and no edges, is formalized as follows: $pg_\emptyset$ with $N_{pg_\emptyset} = E_{pg_\emptyset} = \emptyset$, $src_{pg_\emptyset} = dest_{pg_\emptyset} = labels_{pg_\emptyset} = \emptyset \to \emptyset$ and $properties_{pg_\emptyset} : \emptyset \times \emptyset \to \emptyset$.

3.3.1. Renaming Property Graphs and isomorphism

The chosen formal definition of the running example is not the only one that is possible: for example an arbitrary element named a could have been used in place of n_1 as the first listed node identifier in Example 3.

Definition 5 (Renaming function)**.** For all sets N_1, N_2, E_1, E_2 where $N_1 \cap E_1 = \emptyset$ and $N_2 \cap E_2 = \emptyset$, a renaming from N_1 and E_1 to N_2 and E_2 is a bijective function $\phi : N_1 \cup E_1 \to N_2 \cup E_2$ where $\forall n \in N_1, \phi(n) \in N_2 \wedge \forall e \in E_1$, $\phi(e) \in E_2$.

Example 4. Let TT' be another formal definition of the PG in Fig. 1 that uses the sets $N_{TT'} = \{a, b\}$ and $E_{TT'} = \{c\}$.

An example of a renaming function ϕ_{TT} from $N_{TT} = \{n_1, n_2\} \cup E_{TT} = \{e_1\}$ to $N_{TT'} = \{a, b\} \cup E_{TT'} = \{c\}$ is $\phi_{TT} = \{n_1 \mapsto a; n_2 \mapsto b; e_1 \mapsto c\}$.

Definition 6 (Property Graph renaming)**.** Let pg be a property graph, N' and E' be two sets and ϕ be a renaming function from N_{pg} and E_{pg} to N' and E'. We define the renamed PG $rename(\phi, pg) = pg'$ as follows:

- $N_{pg'} = N' = \{\phi(n) | n \in N_{pg}\}$
- $E_{pg'} = E' = \{\phi(e) | e \in E_{pg}\}$
- $src_{pg'} = \{e \mapsto \phi(src_{pg}(\phi^{-1}(e))) | e \in E_{pg'}\}$
- $dest_{pg'} = \{e \mapsto \phi(dest_{pg}(\phi^{-1}(e))) | e \in E_{pg'}\}$
- $labels_{pg'} = \{m \mapsto labels_{pg}(\phi^{-1}(m)) | m \in N_{pg'} \cup E_{pg'}\}$
- $properties_{pg'} = \{(m, key) \mapsto properties_{pg}(\phi^{-1}(m), key) | m \in N_{pg'} \cup E_{pg'} \wedge key \in Str\}$

Example 5. Let us consider back TT, the PG about Tintin defined in Example 3, TT' the other formalization of the same PG and ϕ_{TT} the renaming function introduced in the Example 4.

The PG produced by $rename(\phi_{TT}, TT) = TT'$ is

- $N_{TT'} = \{a, b\}$; $E_{TT'} = \{c\}$
- $src_{TT'} = \{c \mapsto a\}$; $dest_{TT'} = \{c \mapsto b\}$
- $labels_{TT'} = \{a \mapsto \{\text{“}Person\text{”}\}; b \mapsto \emptyset; c \mapsto \{\text{“}TravelsWith\text{”}\}\}$
-

$$properties_{TT'} = \left\{ \begin{array}{c} (a, \text{“}name\text{”}) \mapsto \text{“}Tintin\text{”}; (a, \text{“}job\text{”}) \mapsto \text{“}Reporter\text{”} \\ (b, \text{“}name\text{”}) \mapsto \text{“}Snowy\text{”}; (c, \text{“}since\text{”}) \mapsto 1978 \end{array} \right\}$$

Definition 7 (Isomorphic property graph)**.** Two PGs pg and pg' are isomorphic iff there exists a renaming function ϕ from N_{pg} and E_{pg} to $N_{pg'}$ and $E_{pg'}$ such that $rename(\phi, pg) = pg'$. Note that because renaming functions are bijectives, it implies that ϕ^{-1} exists and $rename(\phi^{-1}, pg') = pg$.

Note that both TT and TT' from Examples 3 and 4 match the graphical representation given in Fig. 1. An informal way to define the isomorphism between two PGs is to check if they have the same graphical representation.

Existing works [16,33] on PG query languages focus on extracting the properties of some nodes and edges, and never look for the exact identity of the elements. It is therefore possible to affirm that the exact identity is not important, and that if two PGs are isomorphic, they are the same PG for practical matter.

3.4. RDF-star definition

Definition 8 (Atomic RDF terms)**.** Let I be the infinite set of IRIs, $L = Str \times I$ be the set of literals and B be the infinite set of blank nodes. The sets I, L and B are disjoint.

IRIs, literals and blank nodes are grouped under the name “Atomic RDF terms”.

Notation: In the examples, the IRIs, the elements of I, will be either noted as full IRIs between brackets, *e.g.* `<http://example.org/Tintin>` or by using prefixes to shorten the IRI *e.g.* `ex:Tintin`. The list of prefixes used in this paper is described in Table 2.

Literals, the elements of L, can be noted either by using the usual tuple notation, *e.g.* (“*1978*”, `xsd:integer`) or with the compact notation “*1978*”$^{xsd:integer}$.

Finally, the blank nodes, the elements of B, are denoted by blank node labels prefixed with the two symbols “_:” *e.g.* _:*edge*, _:*2021* or _:*node35*.

Definition 9 (RDF(-star) triples and graphs)**.** The set of all RDF triples[6] is denoted *RdfTriples* and is defined as follows:

- $\forall subject \in I \cup B, \forall predicate \in I, \forall object \in I \cup B \cup L, (subject, predicate, object) \in RdfTriples$.
- $\forall tsubject \in RdfTriples, \forall tobject \in RdfTriples$, and for all *subject*, *predicate* and *object* defined as above, (*tsubject*, *predicate*, *object*), (*subject*, *predicate*, *tobject*) and (*tsubject*, *predicate*, *tobject*) are members of *RdfTriples*.

Table 2
List of prefixes used in this paper

Prefix	IRI
rdf	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
xsd	`http://www.w3.org/2001/XMLSchema#`
ex	`http://example.org/`
prec	`http://bruy.at/prec#`
pvar	`http://bruy.at/prec-var#`

A subset of *RdfTriples* is an RDF graph.

Both the atomic RDF terms defined in Definition 8 and RDF triples are *terms*. A triple used in another triple, in subject or object position, is a *quoted triple*. An RDF triple can not contain itself and can not be nested infinitely.

Example 6.

1. The triple (*ex*:*tintin*, *rdf*:*type*, *ex*:*Person*) is an element of *RdfTriples*. Its Turtle representation is `ex:tintin rdf:type ex:Person.`
2. The RDF graph exposed in Listing 1 is composed of 5 triples written in Turtle format. In our formalism, the second triple, `_:tintin foaf:name "Tintin"`, is (_:*tintin*, *foaf*:*name*, "*Tintin*"$^{xsd:string}$).
3. (*ex*:*tintin*, *ex*:*travelsWith*, *ex*:*snowy*) is an element of *RdfTriples*.
((*ex*:*tintin*, *ex*:*travelsWith*, *ex*:*snowy*), *ex*:*since*, "1978"$^{xsd:integer}$) is an element of *RdfTriples* that has a quoted triple in subject position.

Definition 10 (Term membership)**.** The $\in$ operator is extended to triples to check if a term is part of a triple.

$$\forall term \in I \cup B \cup L \cup RdfTriples, \forall (s, p, o) \in RdfTriples, term \in (s, p, o)$$

$$\Leftrightarrow \begin{bmatrix} & term = s \\ \vee & (s \in RdfTriples \wedge term \in s) \\ \vee & term = p \\ \vee & term = o \\ \vee & (o \in RdfTriples \wedge term \in o) \end{bmatrix}$$

Example 7 (Term membership examples)**.**

- *rdf*:*type* $\in$ (*ex*:*tintin*, *rdf*:*type*, *ex*:*Person*).
- *ex*:*snowy* $\notin$ (*ex*:*tintin*, *rdf*:*type*, *ex*:*Person*).
- _:*n* $\in$ (_:*n*, *rdf*:*type*, *ex*:*Person*)
- _:*e* $\notin$ (_:*n*, *rdf*:*type*, *ex*:*Person*)
- *xsd*:*string* $\in$ (*xsd*:*string*, *ex*:*p*, *ex*:*o*)
- *xsd*:*string* $\notin$ (*ex*:*tintin*, *ex*:*name*, "*Tintin*"$^{xsd:string}$)
- *ex*:*tintin* $\in$ ((*ex*:*tintin*, *ex*:*travelsWith*, *ex*:*snowy*), *ex*:*since*, "1978"$^{xsd:integer}$)

Definition 11 (List of blank nodes used in an RDF graph)**.** For every RDF graph *rdf*, *BNodes*(*rdf*) is the set of blank nodes in *rdf i.e.* $\forall rdf \subseteq RdfTriples, BNodes(rdf) = \{bn \in B | \exists t \in rdf, bn \in t\}$.

Example 8. Let *GTT* be the RDF graph exposed on Listing 1. *BNodes*(*GTT*) = {_:*tintin*, _:*snowy*}

4. PRSC: Mapping PGs to RDF graphs

PRSC enables the user to convert any Property Graph to an RDF graph by using user-defined templates.

[6]For the sake of readability, although RDF-star is not yet part of the official RDF recommendation [34], we conflate RDF-star and RDF in this paper. When we mention an RDF triple or an RDF graph, we allow them to contain quoted triples.

4.1. Property graphs with blank nodes

By default, every node and edge of the PG to convert is mapped by PRSC to an arbitrary fresh blank node. This mapping is arbitrary because blank nodes are essentially interchangeable, and because they have no global identifiers which would allow to map to a specific blank node anyway. This mapping is used throughout the transformation. For the sake of conciseness, we introduce the following modelling trick.

We note that in the respective definitions of PGs and RDF, the sets N and E of nodes and edges (in any PG) and the global set B of blank nodes (in RDF), are very loosely characterized. The only constraints are that N and E are disjoint and finite, and that B is disjoint from the sets of IRIs and literals. Theoretically, nothing prevents a property graph to take its nodes and edges in the set B, in other words, to have $N \subset B$ and $E \subset B$.

Our arbitrary mapping from $N \cup E$ to fresh blank nodes can then be used as a renaming function to build a new PG that is isomorphic to the original one. As noted in Section 3.3.1, the two PGs are indistinguishable for any practical purpose, including the transformation to an RDF graph. Without loss of generality, all the definitions and algorithms in this paper expect PGs whose nodes and edges are elements of B. This amounts to assume that the arbitrary mapping to blank nodes has been set in advance, and is carried by the PG to transform: every element of $N \cup E$ is identified with the blank nodes it maps to.

Conversely, when we study the reversibility of some contexts in Section 5, we prove that the produced BPG is exactly the original one. Producing a PG with arbitrary elements, and proving that it was isomorphic to the original, would be much more complex.

Definition 12 (Blank Node Property Graph)**.** *BPGs* is the set of property graphs with blank nodes only (BPG), *i.e.* $BPGs = \{pg \in PGs | (N_{pg} \cup E_{pg}) \subseteq B\}$.

Example 9. By defining the renaming function $\phi_{BTT} = \{n_1 \mapsto \_{:}n1; n_2 \mapsto \_{:}n2; e_1 \mapsto \_{:}e1\}$, it is possible to build the BPG $BTT = rename(\phi_{BTT}, TT)$:

- $N_{BTT} = \{\_{:}n1, \_{:}n2\}$; $E_{BTT} = \{\_{:}e1\}$
- $src_{BTT} = \{\_{:}e1 \mapsto \_{:}n1\}$; $dest_{TT'} = \{\_{:}e1 \mapsto \_{:}n2\}$
- $labels_{BTT} = \{\_{:}n1 \mapsto \{\text{“}Person\text{”}\}; \_{:}n2 \mapsto \emptyset; \_{:}e1 \mapsto \{\text{“}TravelsWith\text{”}\}\}$
-

$$properties_{BTT} = \left\{ \begin{array}{c} (\_{:}n1, \text{“}name\text{”}) \mapsto \text{“}Tintin\text{”}; (\_{:}n1, \text{“}job\text{”}) \mapsto \text{“}Reporter\text{”} \\ (\_{:}n2, \text{“}name\text{”}) \mapsto \text{“}Snowy\text{”}; (\_{:}e1, \text{“}since\text{”}) \mapsto 1978 \end{array} \right\}$$

By construction, BTT is isomorphic to TT, and as $N_{BTT} \cup E_{BTT} \subseteq B$, $BTT \in BPGs$.

This may raise the question of the meaning of two BPGs that share the same blank nodes, in particular if one blank node is used as a node in one BGP and as an edge in the other. However, this question could also be raised for all PGs: what would it mean for two PGs to have common elements in their respective sets of nodes and edges? In general, it would not hold any semantics, as PG elements are considered locally for a given PG. However, in Section 5.3.4, we will define operators for PGs, even the ones without blank nodes, that will consider PGs with shared PG elements and their possible relations.

4.2. Type of a PG element and PG schemas

We define the type of a PG element and PG schemas as follows. Let pg be a PG.

Definition 13 (Property keys of an element)**.** We recall that in Section 2 and in Definition 3, properties are described as key-value pairs.

$keys_{pg}$ is the function that maps a PG element (node or edge) of the PG pg to the list of property keys for which it has a value, *i.e.* $keys_{pg} : N_{pg} \cup E_{pg} \rightarrow 2^{Str}$, with $\forall m \in (N_{pg} \cup E_{pg})$, $keys_{pg}(m) = \{key | properties_{pg}(m, key) \text{ is defined}\}$.

Table 3
The types of the elements in the PG *BTT*

m	$typeof_{BTT}(m)$
_:n1	(“*node*”, {“*Person*”}, {“*name*”, “*job*”})
_:n2	(“*node*”, ∅, {“*name*”})
_:e1	(“*edge*”, {“*TravelsWith*”}, {“*since*”})

Definition 14 (Type of a PG element)**.** A type is a triple composed of 1) the *kind* of the PG element, *i.e.* if it is a node or an edge, 2) a set of labels and 3) a set of property keys. The set of all types is denoted *Types* and is defined as $Types = (\{\text{“}node\text{”}, \text{“}edge\text{”}\} \times 2^{Str} \times 2^{Str})$.

The type of an element $m \in N_{pg} \cup E_{pg}$ is

$$typeof_{pg}(m) = \left(\begin{cases} \text{“}node\text{”} & \text{if } m \in N_{pg} \\ \text{“}edge\text{”} & \text{if } m \in E_{pg} \end{cases}, labels_{pg}(m), keys_{pg}(m) \right)$$

A set of PG types is named a *schema*.

The functions *kind*, *labels* and *keys* are defined for types such that $\forall type = (u, l, key) \in Types$, $kind(type) = u$, $labels(type) = l$, $keys(type) = key$.

Example 10. Table 3 shows the types of the PG elements in the running example.

Remark 2. If two PGs pg and pg' are isomorphic, their elements share the same type.

Indeed, by definition, there exists a renaming function ϕ from the elements of pg to the elements of pg', and $\forall m \in N_{pg} \cup E_{pg}$, $typeof_{pg}(m) = typeof_{pg'}(\phi(m))$.

4.3. Template triples

PRSC resorts to a mechanism of templating: to produce an RDF graph from a PG, we use tuples of three elements, named template triples, that will be mapped to proper RDF triples.

Definition 15 (Placeholders)**.** There are four distinct elements, not included in either of the previously defined sets, named *valueOf*, ?*self*, ?*source* and ?*destination*.[7]

Let $pvars = \{?self, ?source, ?destination\}$. *pvars* elements serve as placeholders that will be replaced by the blank nodes that represent the nodes and edges in the PG.

Let $P = \{(l, valueOf) | l \in Str\}$. Elements of P can be noted with the same syntax as literals, for example $\text{“}name\text{”}^{valueOf}$ is the pair (“*name*”, *valueOf*). Each element of P serves as a placeholder to be replaced with an RDF literal that represents the value of a property in the PG.

Definition 16 (Template triples)**.** A *template triple* is a member of *Templates* and is defined as follows:

- $\forall subject \in I \cup pvars, \forall predicate \in I, \forall object \in I \cup pvars \cup L \cup P$, $(subject, predicate, object) \in Templates$.
- $\forall tsubject \in Templates$, $\forall tobject \in Templates$, and for all *subject*, *predicate* and *object* defined as above, $(tsubject, predicate, object)$, $(subject, predicate, tobject)$ and $(tsubject, predicate, tobject)$ are members of *Templates*.

Note that unlike RDF triples, the elements of *Templates* can not contain blank nodes but can contain placeholders: *pvars* members can be used in subject (first) and/or object (third) position as they will be mapped later to blank nodes, and members of P are allowed in object position as they will be mapped later to literals. Similarly to RDF triples, template triples can not contain themselves and can not be nested infinitely.

Any subset of *Templates* is named a **template graph**. The PRSC engine will use template graphs to produce RDF graphs.

[7] In practice, our implementation of this paper maps *valueOf* to the IRI `prec:valueOf` and all terms prefixed with ? to the `pvar` namespace. Examples given in Turtle reflect the implementation instead of fully fitting the theoretical definitions.

Example 11. The triple (*ex*:*tintin*, *rdf*:*type*, *ex*:*Person*) is both an element of *RdfTriples* and an element of *Templates*. Indeed, both *RdfTriples* and *Templates* allow IRIs in the subject, predicate and object positions.

The triple (?*self*, *rdf*:*type*, *ex*:*Person*) is a member of *Templates* but not of *RdfTriples*. The first member is an element of *pvars*, which is only allowed in template triples. The two other members are IRIs, and $pvars \times I \times I$ is a subset of *Templates*.

The triple (*ex*:*tintin*, *ex*:*name*, "*name*"^{valueOf}) is also a member of *Templates* but not of *RdfTriples*. While the first two members are IRIs, the third member is an element of P. Again, elements of P are only allowed by template triples.

Definition 17 (Placeholders and template triple membership)**.** The $\in$ operator, which we extended in Definition 10, is further extended to placeholders and template triples:

$$\forall term \in I \cup B \cup L \cup RdfTriples \cup Templates, \forall (s, p, o) \in RdfTriples \cup Templates,$$

$$term \in (s, p, o) \Leftrightarrow \begin{bmatrix} & term = s \\ \vee & (s \in RdfTriples \cup Templates \wedge term \in s) \\ \vee & term = p \\ \vee & term = o \\ \vee & (o \in RdfTriples \cup Templates \wedge term \in o) \end{bmatrix}$$

4.4. PRSC context

In this paper, the notion of PRSC context is the keystone to let the user drive the conversion from a PG to an RDF graph. It maps PG types to template graphs. The *prsc* algorithm proceeds by looping on each node and edge of the PG, computing its type, finding the associated template graph in the context, and replacing the placeholders of this template graph with data extracted from the PG to produce an RDF graph.

Definition 18 (PRSC Context)**.** A PRSC context $ctx : Types \rightarrow 2^{Templates}$ is a partial function that maps types to template graphs.

All template graphs must be *valid*, *i.e.* for all types, the placeholders used in the associated template graph must be consistent with the type: (1) for any given property key, its associated placeholder may only be used in template graphs associated with types that contain the property key, for example the placeholder "*name*"^{valueOf} may only be used if the property key "*name*" is in the type associated to this template; (2) and templates associated to node types are not allowed to use the placeholders ?*source* and ?*destination*, as they are related to the source or the destination of an edge.

Formally, all template graphs used by a context *ctx* are valid iff $\forall type \in Dom(ctx)$:

1. $\forall (key, valueOf) \in P, (\exists tp \in ctx(type) | (key, valueOf) \in tp) \Rightarrow key \in keys(type)$.
2. $(kind(type) = \text{"}node\text{"}) \Rightarrow [\nexists tp \in ctx(type) | ?source \in tp \vee ?destination \in tp]$.

The set of all context functions is denoted *Ctx*.

Definition 19 (Complete PRSC contexts for a given PG)**.** A PRSC context is said complete for a property graph $pg \in BPGs$ iff there is a template graph defined for each type used in *pg*. The set of all complete contexts for a PG *pg* is noted $Ctx_{pg} = \{ctx \in Ctx | \forall m \in N_{pg} \cup E_{pg}, typeof_{pg}(m) \in Dom(ctx)\}$.

Remark 3. Note that the type system described in Definition 14 is trivial to resolve as the type of a PG element *m*, denoted by $typeof_{pg}(m)$, only depends on its kind (node or edge), the list of its labels and the list of its property keys. For this reason, checking if a PRSC context *ctx* is complete for a property graph *pg* is also trivial as it consists in computing the type of each PG element of *pg* and checking if all types are in the domain of *ctx*.

Example 12. Table 4 exposes an example of a complete *ctx* context function for our running example. First, the function is a context as all template graphs are valid: The placeholders "*name*"^{valueOf} and "*job*"^{valueOf} are only used in types with the associated property key. The fact that the property key "*since*" in the third type has no associated placeholder occurrence in the template graph does not invalidate the context. The placeholders ?*source*

Table 4
An example of a complete context for the Tintin Property Graph

type	*ctx*(*type*)
(“*node*”, {“*Person*”}, {“*name*”, “*job*”})	{(?*self*, *rdf*:*type*, *ex*:*Person*), (?*self*, *foaf*:*name*, “*name*”^{valueOf}), (?*self*, *ex*:*profession*, “*job*”^{valueOf})}
(“*node*”, ∅, {“*name*”})	{(?*self*, *foaf*:*name*, “*name*”^{valueOf})}
(“*edge*”, {“*TravelsWith*”}, {“*since*”})	{(?*source*, *ex*:*isTeammateOf*, ?*destination*)}

Table 5
An incomplete context for the Tintin PG

type	*ctx*(*type*)
(“*node*”, {“*Person*”}, {“*name*”, “*job*”})	{(?*self*, *rdf*:*type*, *ex*:*Person*), (?*self*, *foaf*:*name*, “*name*”^{valueOf}), (?*self*, *ex*:*profession*, “*job*”^{valueOf})}

Table 6
A function that is not a context

type	*ctx*(*type*)
(“*node*”, {“*Person*”}, {“*name*”, “*job*”})	{(?*self*, *ex*:*familyName*, “*surname*”^{valueOf})}
(“*node*”, ∅, {“*name*”})	{(?*self*, *foaf*:*name*, “*name*”^{valueOf})}
(“*edge*”, {“*TravelsWith*”}, {“*since*”})	{(?*source*, *ex*:*isTeammateOf*, ?*destination*)}

and ?*destination* are only used in the third type, which is an edge type. Then, all three types used by our running example have an associated template graph, so it is a complete context for the PG exposed in Example 1.

Example 13. The function *ctx* exposed in Table 5 is not complete for the PG *BTT* as its domain lacks the type of _:*n2* and the type of _:*e1*.

Example 14. The function *ctx* exposed in Table 6 is not a context because “*surname*”, which is used in the template graph mapped to the first listed type (“*node*”, {“*Person*”}, {“*name*”, “*job*”}), is not a value in {“*name*”, “*job*”}.

4.5. Application of a PRSC context on a PG

We now define formally the conversion operated by PRSC. A PRSC conversion of a PG depends on a chosen context $ctx \in Ctx$.

Definition 20 (Property value conversion)**.** For the conversion of property values to literals, we consider that we have a fixed total injective function $toLiteral : V \to L$, common for all PGs and contexts. We suppose that *toLiteral* is reversible, *i.e.* we are able to compute $toLiteral^{-1}$.

Definition 21 (The *prsc* function)**.** The operation that produces an RDF graph from the application of a PRSC context $ctx \in Ctx_{pg}$ on a property graph $pg \in BPGs$ is noted $prsc(pg, ctx)$. The result of the *prsc* function is the union of the RDF graph built by converting all elements of the PG, into RDF. The conversion of a single element is materialized by the *build* function.

$\forall tps \subseteq Templates, \forall pg \in BPGs, \forall m \in N_{pg} \cup E_{pg}, build(tps, pg, m) = \{\beta_{pg,m}(tp) | tp \in tps\}$ with $\beta_{pg,m}$ defined as follows:

$$\beta_{pg,m} : \begin{cases} Templates & \to RdfTriples \\ P \cup L & \to L \\ I & \to I \\ pvars & \to B \end{cases}$$

Table 7
Application of a PRSC context on the running example

m	$typeof_{BTT}(m)$	$ctx(typeof_{BTT}(m))$	$build(ctx(typeof_{BTT}(m)), BTT, m)$
_:n1	(“node”, {“Person”}, {“name”, “job”})	{(?self, rdf:type, ex:Person), (?self, foaf:name, “name”^{valueOf}), (?self, ex:profession, “job”^{valueOf})}	{(_:n1, rdf:type, ex:Person), (_:n1, foaf:name, “Tintin”), (_:n1, ex:profession, “Reporter”)}
_:n2	(“node”, ∅, {“name”})	{(?self, foaf:name, “name”^{valueOf})}	{(_:n2, foaf:name, “Snowy”)}
_:e1	(“edge”, {“TravelsWith”}, {“since”})	{(?source, ex:isTeammateOf, ?destination)}	{(_:n1, ex:isTeammateOf, _:n2)}

$$\beta_{pg,m}(x) = \begin{cases} (\beta_{pg,m}(x_s), \beta_{pg,m}(x_p), \beta_{pg,m}(x_o)) & \text{if } x = (x_s, x_p, x_o) \in Templates \\ x & \text{if } x \in L \cup I \\ m & \text{if } x = ?self \\ src_{pg}(m) & \text{if } x = ?source \wedge m \in E_{pg} \\ dest_{pg}(m) & \text{if } x = ?destination \wedge m \in E_{pg} \\ toLiteral(properties_{pg}(m, key)) & \text{if } x = (key, valueOf) \in P \\ \text{undefined} & \text{otherwise} \end{cases}$$

As said previously, the result of *prsc* is the union of the graphs produced by *build*, *i.e.*

$$prsc(pg, ctx) = \bigcup_{m \in N_{pg} \cup E_{pg}} build\big(ctx\big(typeof_{pg}(m)\big), pg, m\big)$$

Example 15. Table 7 exposes the resolution of *prsc* on the running example.

The resolution of _:*n2* is as follows:

$$\begin{aligned}
&build(ctx\big(typeof_{BTT}(\text{_:}n2), BTT, \text{_:}n2\big) \\
&\quad = build\big(ctx\big(\text{“}node\text{”}, \emptyset, \{\text{“}name\text{”}\}\big)\big), BTT, \text{_:}n2\big) && \text{Resolution of the type} \\
&\quad = build\big(\big\{\big(?self, foaf{:}name, \text{“}name\text{”}^{prec:valueOf}\big)\big\}, BTT, \text{_:}n2\big) && \text{Application of } ctx \\
&\quad = \big\{\big(\text{_:}n2, foaf{:}name, toLiteral\big(properties_{BTT}(\text{_:}n_2, \text{“}name\text{”})\big)\big)\big\} && \text{Application of } build \\
&\quad = \big\{\big(\text{_:}n2, foaf{:}name, toLiteral(\text{“}Snowy\text{”})\big)\big\} && \text{Evaluation of the property “name”} \\
&\quad = \big\{\big(\text{_:}n2, foaf{:}name, \text{“}Snowy\text{”}^{xsd:string}\big)\big\} && \text{Application of } toLiteral
\end{aligned}$$

Algorithm 1 gives an algorithmic view of the *prsc* function presented by Definition 21.

4.6. Complexity analysis

In this section, we discuss the different metrics that can be used to evaluate the complexity and evaluate the complexity of the *prsc* function.

4.6.1. Functions considered constant

For a given PG pg, the complexity of the functions src_{pg}, $dest_{pg}$, $labels_{pg}$ and $properties_{pg}$ are considered constant.

The complexity of the functions $keys_{pg}$, the *toLiteral* and the $toLiteral^{-1}$ is also considered constant.

Evaluating if something is a member of a given set, for example if an entity is part of the set N_{pg}, is generally considered to be constant time thanks to hash maps.[8]

[8]Inserting and searching in a hash map is not strictly speaking a constant time operation but has an *amortized* constant complexity, and is linear in the worst case.

Algorithm 1: The *prsc* function

Input: $pg \in BPGs$, $ctx \in Ctx_{pg}$
Output: An RDF graph

Main Function *prsc(pg, ctx)*:
  $rdf \leftarrow \{\}$
  forall *PG element* $m \in N_{pg} \cup E_{pg}$ **do**
    $tps \leftarrow ctx(typeof_{pg}(m))$
    /* *build* function */
    $built \leftarrow \{\}$
    forall $tp \in tps$ **do**
      $built \leftarrow built \cup \{\beta(tp, pg, m)\}$ /* $\beta_{tp,pg}(m)$ is necessarily defined, since $ctx \in Ctx_{pg}$ */
    $rdf \leftarrow rdf \cup built$
  return *rdf*
/* In the formal definition, pg and m are implicitly passed to β */
Function $\beta(tp, pg, m)$:
  if $tp \in Templates$ **then**
    $(s, p, o) \leftarrow tp$
    return $(\beta(s, pg, m), \beta(p, pg, m), \beta(o, pg, m))$
  else if $tp \in L$ **then return** tp
  else if $tp \in I$ **then return** tp
  else if $tp \in P$ **then**
    $(key, valueOf) \leftarrow tp$
    return $toLiteral(properties_{pg}(m, key))$
  else if $tp \in pvars$ **then**
    if $tp = ?self$ **then return** m
    if $m \in E_{pg}$ **then**
      switch tp **do**
        case ?*source* **do return** $src_{pg}(m)$
        case ?*destination* **do return** $dest_{pg}(m)$
  /* $\beta_{tp,pg}(m)$ is undefined according to Definition 21 */
  raise *Error(Undefined behavior)*

4.6.2. Considered metrics

For a given PG pg and a given context ctx, the following metrics are considered:

- The number of nodes and edges in pg, denoted *NbOfPGElements*.

$$NbOfPGElements = |N_{pg} \cup E_{pg}|$$

- The size of the biggest template graph, denoted *BiggestTemplateSize*.

$$BiggestTemplateSize = \max_{type \in Dom(ctx)} \left(\left| ctx(type) \right| \right)$$

- The complexity of the types in the context, denoted *TypeComplexity*, reflected by the number of labels and the number of properties of the type with the highest number of labels and properties. Note that since the context has to be valid, *i.e.* all elements of the PG must have their type in the context, *TypeComplexity* is also an upper

bound of the type complexity of the types in the PG.

$$TypeComplexity = 1 + \max_{type \in Dom(ctx)} \left(\left| labels_{pg}(type) \right| + \left| keys_{pg}(type) \right| \right)$$

– The number of types supported by the context $NbTypes = |Dom(ctx)|$.

In RDF-star, quoted triples can be used as subject or object of other triples, without limit on how deeply triples can be nested. In practice, however, it is rare to have more than one level of nesting. Usually, users are expected to use atomic RDF triples like `_:tintin :travelsWith _:haddock` or to use RDF-star triples with a depth of one like `<< _:tintin :travelsWith _:haddock >> :since 1978`. We therefore consider the *depth* of any triple to be bound by a constant. As a consequence, in all functions processing terms and triples recursively (such as β in Algorithm 1), we can ignore the recursion depth in the complexity analysis.

In all complexity analyses, all metrics are considered non null. Indeed, if there are no elements or if the biggest template graph is empty, the produced RDF graph will be empty so this case is not interesting. As we add one to the number of labels and properties, the type complexity can never be zero, even if the context only supports nodes and edges with no labels and no properties.

4.6.3. Complexity of ctx calls in the prsc function

For each given PG element m, the complexity of a call to $ctx(typeof_{pg}(m))$ is $\mathcal{O}(TypeComplexity * \ln(TypeComplexity))$:

– The type of m in the PG pg must be computed. $typeof_{pg}(m)$ has a complexity of $\mathcal{O}(1)$:

  * Evaluating if m is a node or an edge is constant as checking if an element is a member of a set is constant.
  * Calls to $labels_{pg}(m)$ and $keys_{pg}(m)$ are considered constants in Section 4.6.1.

– In the complexity analysis, we consider that *ctx* is implemented as a hash map from the types to the template graphs. A *ctx* call would first need to compute the hash of the type. To do so, it has to look at all the labels and properties in the type in a deterministic order; for this, the labels and keys needs to be sorted, which has a complexity of $\mathcal{O}(TypeComplexity * \ln(TypeComplexity))$. After the hash has been computed, the cost of retrieving the template graph has an amortized constant complexity. The overall complexity of a *ctx* call is $\mathcal{O}(TypeComplexity * \ln(TypeComplexity))$.

4.6.4. Complexity of prsc

Given a PG $pg \in BPGs$ and a context $ctx \in Ctx_{pg}$,

– Calls to $ctx(typeof_{pg}(m))$ in line 4 have a complexity of $\mathcal{O}(TypeComplexity * \ln(TypeComplexity))$.
– Calls to the β function in line 7 are constant, as it has been assumed that the depth of the most nested triple is low enough to be ignored and the operations it performs are in constant time.
– There are two for loops, one iterating on all PG elements (*NbOfPGElements*) and one iterating on all template triples of a template graph (*BiggestTemplateSize*). All instructions in the *prsc* but the one on line 4 are computed in constant time. Line 4 is inside the first loop but outside the second loop.

The *prsc* function has an $\mathcal{O}(NbOfPGElements * (BiggestTemplateSize + TypeComplexity * \ln(TypeComplexity)))$ complexity.

5. PRSC reversibility

When PGs are converted into RDF graphs, an often desired property is to not have any information loss. To determine whenever or not a conversion induces information loss is to check if the conversion is reversible, *i.e.* if from the output, it is possible to compute back the input. The reversion is studied relatively to the used PRSC contexts: the PRSC context is used as both an input of both the PRSC algorithm and the reversion algorithm. In other words, we consider that the information stored in the PRSC context do not need to be stored in the produced RDF graph to produce a reversible conversion.

This section first shows that not all PRSC contexts are reversible. Then, properties are exhibited about PRSC contexts, leading to a description of a subset of reversible PRSC contexts, *i.e.* contexts that we prove do not induce information loss.

5.1. Reversibility in this paper

In this paper, we call a function f reversible if we can find back x in practice from $f(x)$. This implies that:

- The function f must be injective. Indeed, if two different values x and x' can produce the same value y, it is impossible to know if the value responsible for producing y was x or x'.
- The inverse function f^{-1} must be computable and tractable in reasonable time. By counter-example, a public-key encryption function is supposed to be injective. It is theoretically possible, although prohibitively costly, to decipher a given message by applying the encryption function on all possible inputs until the result is the original encrypted message. This is not the kind of "reversibility" we are interested in.

We say that a context *ctx* is reversible if for any PG $pg \in BPGs$ such that the context *ctx* is complete for the PG *pg*, it is possible to find back *pg* from the context *ctx* and the result of $prsc(pg, ctx)$.

More formally, when studying reversibility, we want to check if for a given $ctx \in Ctx$, we are able to define a tractable function ${prsc_{ctx}}^{-1}$ such that $\forall pg \in BPGs, [ctx \in Ctx_{pg} \Rightarrow prsc_{ctx}^{-1}(prsc(pg, ctx)) = pg]$.

Example 16 (A trivially non-reversible context)**.** Consider the context $ctx_\emptyset$ such that for all types, it returns the empty template graph, *i.e.* $\forall type \in Types, ctx_\emptyset(type) = \emptyset$. As it is complete for all property graphs, it is possible to use this context on any property graph. However, applying the context $ctx_\emptyset$ produces the empty RDF graph. Therefore, the use of the context $ctx_\emptyset$ makes the function *prsc* not injective, and therefore not reversible.

Example 17 (A more realistic example of a non-reversible context)**.** Another example of a non-reversible context is the context exposed in Table 4: while this context can be applied on PGs in which edges have the "since" property, the value of this property will never appear in the produced RDF graph.

As not all contexts are reversible, the next sections focus on characterizing some contexts that produce reversible conversions.

5.2. Well-behaved contexts

5.2.1. Characterization function

To be able to reverse back to the original PG, we need a way to distinguish the triples that may have been produced by a given member of *Templates* from the ones that cannot have been produced by it. For this purpose, this section introduces the κ function. This function must verify that, for every triple template *tp* and every triple t, $\kappa(t) = \kappa(tp)$ if and only if t can be produced from *tp* by the β function. It would then follow that two template triples that may produce the same triple have the same image through κ.

Definition 22 (Characterization function)**.** The κ function maps:

- All template triples to a super set of triples that it is able to generate.
- All RDF triples t to a super-set of the RDF triples that a template triple that may generate the triple t may also generate. For example, a literal may be generated by any element of P. An element of P may generate any literal. Therefore, the κ function maps all literals to the set of all literals.

$$\kappa : \begin{cases} Templates \cup RdfTriples & \rightarrow 2^{RdfTriples} \\ L \cup P & \rightarrow \{L\} \\ I & \rightarrow 2^{I} \\ B \cup pvars & \rightarrow \{B\} \end{cases}$$

Table 8
The running example context with the corresponding values through κ

type	*ctx*(*type*)	$\kappa(ctx(type))$
$tn1 = (\text{“}node\text{”}, \{\text{“}Person\text{”}\}, \{\text{“}name\text{”}, \text{“}job\text{”}\})$	$\{(?self, rdf{:}type, ex{:}Person),$	$(B \times \{rdf{:}type\} \times \{ex{:}Person\})$
	$(?self, foaf{:}name, \text{“}name\text{”}^{valueOf}),$	$\bigcup(B \times \{foaf{:}name\} \times L)$
	$(?self, ex{:}profession, \text{“}job\text{”}^{valueOf})\}$	$\bigcup(B \times \{ex{:}profession\} \times L)$
$tn2 = (\text{“}node\text{”}, \emptyset, \{\text{“}name\text{”}\})$	$\{(?self, foaf{:}name, \text{“}name\text{”}^{valueOf})\}$	$(B \times \{foaf{:}name\} \times L)$
$te1 = (\text{“}edge\text{”}, \{\text{“}TravelsWith\text{”}\}, \{\text{“}since\text{”}\})$	$\{(?source, :isTeammateOf, ?destination)\}$	$(B \times \{:isTeammateOf\} \times B)$

$$\kappa(x) = \begin{cases} \kappa(s) \times \kappa(p) \times \kappa(o) & \text{if } x = (s, p, o) \in RdfTriples \cup Templates \\ L & \text{if } x \in L \cup P \\ \{x\} & \text{if } x \in I \\ B & \text{if } x \in B \cup pvars \end{cases}$$

The κ function is extended to all template graphs and RDF graphs xs as $\kappa(xs) = \bigcup_{t \in xs} \kappa(t)$.

Example 18 (κ applied to the running example from Fig. 1)**.**

- $\kappa(?source) = B, \kappa(\_{:}n1) = B.$
- $\kappa(foaf{:}name) = \{foaf{:}name\}.$
- $\kappa(\text{“}name\text{”}^{valueOf}) = L, \kappa(\text{“}Tintin\text{”}) = L.$
- $\kappa((?self, foaf{:}name, \text{“}name\text{”}^{valueOf})) = B \times \{foaf{:}name\} \times L.$
- $\kappa((\_{:}n1, foaf{:}name, \text{“}Tintin\text{”})) = B \times \{foaf{:}name\} \times L.$
- Note that
  - $\kappa((\_{:}n1, foaf{:}name, \text{“}Tintin\text{”})) = \kappa((?self, foaf{:}name, \text{“}name\text{”}^{valueOf}))$
  - $(\_{:}n1, foaf{:}name, \text{“}Tintin\text{”}) \in \kappa((?self, foaf{:}name, \text{“}name\text{”}^{valueOf}))$
- $\kappa((?source, ex{:}isTeammateOf, ?destination)) = B \times \{ex{:}isTeammateOf\} \times B$
- $\kappa(((?source, ex{:}isTeammateOf, ?destination), ex{:}since, \text{“}since\text{”}^{valueOf})) = (B \times \{ex{:}isTeammateOf\} \times B) \times \{ex{:}since\} \times L$

Table 8 provides an example of applying κ on the running example context of Table 4.

Remark 4 (κ on terms and triples is, as expected, a super-set of the possible generated values)**.** When comparing the definition of the κ function with the β functions defined in Section 4.5, it appears that:

- For elements in B, *pvars*, L and P, the image of κ is equal to the corresponding image set of the β function.
- For elements in I, the image of κ is equal to a singleton containing that element; β maps any IRI to itself.
- If the given term is a triple, the image of κ is the cross product of the application of the κ function to the terms that compose the RDF triple. As β on triples recursively applies itself to the three terms in the triple, we can see that $\forall\beta, \forall triple, \beta(triple) \in \kappa(triple)$.

Therefore, **if** x **is a term or an RDF triple, for any** β **function,** $\beta(x) \in \kappa(x)$.

Remark 5 (The result of *build* is, as expected, a subset of the result of κ)**.** The *build* function, from which *prsc* is defined, uses β on each template triple. After β is applied, the union of the singletons containing each triple is computed. This is similar to the definition of κ on a set of triples.

From Remark 4, it can be deduced that **if** *tps* **is a set of template triples,** $\forall pg, \forall m, build(tps, pg, m) \subseteq \kappa(tps)$.

Remark 6 (A template and its produced values share the same image through κ)**.** When using the κ function, elements in B and *pvars* both map to B, and elements in L and P both map to L. Elements in I are wrapped into a singleton and both *RdfTriples* and *Templates* apply the function recursively on their members.

When using the β function:

- Elements in *pvars* map for all PGs $pg \in BPGs$ to elements of N_{pg} and E_{pg}, which are both subsets of B.
- Elements in P map to elements in Img(*toLiteral*), which is a subset of L.
- Elements in L and I are mapped to themselves.
- Elements in *Templates* apply the β function recursively on their members.

Therefore, $\forall tp \in Templates, \kappa(\beta(tp)) = \kappa(tp)$

As mentioned previously, the role of κ is to allow us to determine whether two template triples with placeholders may produce the same triple. It maps all placeholders to a super-set [9] of all elements they can generate with the *build* function. All RDF Triples are mapped by the κ function to a subset of *RdfTriples* they are a member of.

Lemma 1. *If a triple is generated by a template graph, then there exists a template triple with the same image through* κ.

$\forall pg \in BPGs, \forall m \in (N_{pg} \cup E_{pg}), \forall tps \subseteq Templates, \forall td \in build(tps, pg, m), \exists tp \in tps | \kappa(td) = \kappa(tp)$

Proof. Per the Definition 21 of *build*, a triple can only be generated by a template graph by the application of β to one of its template triples. Per Remark 6, the generated triple and the corresponding template triple have the same image though κ. □

Definition 23 (*unique* template triple)**.** A template triple *tp* is *unique* in a set of template triples if no other template triple in the set has the same image through κ equal to *tp*.

It is defined as follows with $tp \in tps \subset Templates$:

$$unique(tp, tps) = \left(\forall tp' \in tps, \kappa(tp) = \kappa\left(tp'\right) \Leftrightarrow tp = tp'\right)$$

Combined with Remark 6, what $unique(tp, tps)$ tells us is that any triple, with the same image through κ as *tp*, can not have be generated by any other element of *tps* than *tp* itself. This leads us to Theorem 1 below.

Theorem 1 (Triples produced by a *unique* template triple)**.** *In the result of the build function, if a data triple and a* unique *template triple have the same value through* κ*, then the data triple must have been produced by this template triple.*

$\forall pg \in BPGs, \forall ctx \in Ctx_{pg}, \forall m \in (N_{pg} \cup E_{pg})$, *let* $tps = ctx(typeof_{pg}(m)), \forall td \in build(tps, pg, m), \forall tp \in tps$*:*

$$unique(tp, tps) \wedge \kappa(td) = \kappa(tp) \Rightarrow td \in build\left(\{tp\}, pg, m\right)$$

Proof. We prove the theorem by contradiction.

Let us suppose that:

- (A) $td \in build(tps, pg, b)$
- (B1) $unique(tp, tps)$, i.e. $(\forall tp' \in tps, \kappa(tp) = \kappa(tp') \Rightarrow tp = tp')$

[9] Note that as κ maps to a super set, it may catch false positives. For example, P can only generate elements in *Img*(*toLiteral*), but the κ function considers that all elements of L can be generated from P. For the scope of this paper, κ catching false positives is considered acceptable, as we are only trying to prove the reversibility of a given class of contexts, rather than to characterize the whole class of reversible contexts.

- (B2) $\kappa(td) = \kappa(tp)$
- (C) $td \notin build(\{tp\}, pg, b)$

$$
\begin{aligned}
& td \in build\big(tps - \{tp\}, pg, b\big) && \text{[(A) and (C)]} \\
& \quad \Rightarrow \exists tdp \in tps - \{tp\}, \kappa(tdp) = \kappa(td) && \text{[Lemma 1]} \\
& \quad \Rightarrow \exists tdp \in tps - \{tp\}, \kappa(tdp) = \kappa(tp) && \text{[(B2)]} \\
& \quad \Rightarrow \exists tdp \in tps - \{tp\}, tdp = tp && \text{[(B1)]} \\
& \quad \Rightarrow tp \in tps - \{tp\}
\end{aligned}
$$

tp can not be a member of the set $tps - \{tp\}$, as it explicitly exclude it. As we reached a contradiction, it means that $td \in build(\{tp\}, pg, b)$. □

Theorem 1 allows us to link an RDF triple to the unique template triple that produced it. Then by comparing the terms of the RDF triple to the corresponding placeholders in the template triple, we will be able to reconstruct the original PG.

5.2.2. Well-behaved PRSC context

In this section, we define a subset of contexts that we call *well-behaved PRSC contexts*. In the next section, we will prove that these contexts are reversible.

Definition 24 (Well-behaved contexts)**.** A PRSC context ctx is well-behaved if it conforms to those 3 criteria:
$\forall type \in Dom(ctx)$, let $tps = ctx(type)$

1. *Element provenance*: all generated triples must contain the blank node that identifies the node or the edge it comes from. This is achieved by using the $?self$ placeholder in all template triples:
   - $\forall tp \in tps, ?self \in tp$
2. *Signature template triple*: tps contains at least one template triple, called its *signature* and noted $sign_{ctx}(type)$, that will produce triples that no other template in ctx can produce. This will allow, for each blank node in the produced RDF graph, to identify its type in the original PG.
   - $\exists sign_{ctx}(type) \in tps, \forall x \in Dom(ctx), \kappa(sign_{ctx}(type)) \subseteq \kappa(ctx(x)) \Rightarrow x = type$
3. *No value loss*: for all elements in the PG, we do not want to lose information stored in properties, nor for edges, the source and destination node. Each of these pieces of information must be present in an unambiguously recognizable triple pattern.
   - $\forall key \in keys(type), \exists tp \in tps | unique(tp, tps) \wedge (key, valueOf) \in tp$
   - $kind(type) =$ "$edge$" $\Leftrightarrow \exists tp \in tps | unique(tp, tps) \wedge ?source \in tp$
   - $kind(type) =$ "$edge$" $\Leftrightarrow \exists tp \in tps | unique(tp, tps) \wedge ?destination \in tp$

The set of all well-behaved contexts is Ctx^+, and the set of all well-behaved contexts for a PG pg is Ctx^+_{pg}. $Ctx^+ \subset Ctx$ and $Ctx^+_{pg} = Ctx^+ \cap Ctx_{pg}$.

Remark 7 (Handling multiple $sign_{ctx}$ candidates)**.** In the case where there are multiple template triples candidates to become the signature template triple, the choice of the signature template triple among the candidates is generally not important.

To make the choice deterministic, it could be considered that the chosen signature template triple is the first in lexicographic order. In the case of the presented algorithms, the choice of the signature template triple is not important, and will lead to the same output.

Remark 8 (The template graphs used in well-behaved contexts are not empty)**.** A well-behaved context cannot map a type to an empty template graph: the *signature template triple* criterion ensures that every template graph contains at least one template triple: $\forall tps \in Img(ctx), \exists tp \in tps \Leftrightarrow |tps| \geqslant 1$.

Remark 9 (Inside a well-behaved context, all template graphs are different from all others)**.** For any well-behaved context *ctx*, two types cannot share the same template graph. Indeed, if two types share the same template graph, *i.e.* there are two types *type1* and *type2* with $type1 \neq type2$ such that $ctx(type1) = ctx(type2)$, it would contradict the *signature template triple* criterion as it would lead to $type1 = type2$.

Example 19. Table 8 studies the context used in our running example, exposed in Example 12.

- The type *tn1* matches all criteria of a well-behaved PRSC context:
  - All triples contain ?*self*.
  - At least one template triple is a signature: the image through κ of (?*self*, *rdf*:*type*, *ex*:*Person*) is not contained in the image through κ of other types. It is also the case of (?*self*, *ex*:*profession*, "*job*"^{valueOf}).
  - The properties "*name*" and "*job*" have a *unique* template triple inside $\kappa(ctx(tn1))$.
- The type *tn2* violates the *signature template triple* criterion as (?*self*, *foaf*:*name*, "*name*"^{valueOf}), its only template triple, is shared with the type *tn*1,
- The type *te1* violates the *element provenance* criterion as ?*self* is missing. It also violates the *no value loss* criterion as the term "*since*"^{valueOf} is missing from any template triple.

For all these reasons, this context is not well-behaved.

Example 20 (A well-behaved context for the running example)**.** Let ctx_{TTWB} be the function described in Table 9. In this new context, an arbitrary *ex*:*NamedEntity* IRI is used to sign the PG nodes that have no labels and only a name, and a classic RDF reification is used to model the PG edges.

This context is well-behaved:

- ?*self* appears in all triples,
- Template triples that are signature are marked with a $\star$. At least one signature triple appears for each type,
- All property keys have a *unique* template triple.

Listing 3 is the RDF graph produced by the application of the context ctx_{TTWB} on the PG *BTT*. Each part that starts with a # denotes the application of a *build* application to the PG element described in the comment. The elements are ordered in the same order as their type in Table 9, and the RDF triples and the template triples that produced them are also in the same order.

Remark 10 (Relationship between the empty PG and the empty RDF graph with well-behaved PRSC context)**.** For all well-behaved PRSC contexts, the only PG that can produce the empty RDF graph is the empty PG:

$$\forall pg \in BPGs, ctx \in Ctx^{+}_{pg}, \quad \left|prsc(pg, ctx)\right| = 0 \Leftrightarrow pg = pg_{\emptyset}$$

Indeed, Remark 8 ensures that the template graphs are non-empty. So any application of the *build* function with any well-behaved context produces at least one RDF triple. As the produced RDF graph is the union of the graphs produced by the use of *build* on each node and edge, the only way to have an empty result is to have no node nor edge in the property graph.

Table 9

An example of a complete and well-behaved context for the Tintin Property Graph

type	*ctx*(*type*)
("*node*", {"*Person*"}, {"*name*", "*job*"})	{(?*self*, *rdf*:*type*, *ex*:*Person*)$\star$, (?*self*, *foaf*:*name*, "*name*"^{valueOf}), (?*self*, *ex*:*profession*, "*job*"^{valueOf})$\star$}
("*node*", ∅, {"*name*"})	{(?*self*, *foaf*:*name*, "*name*"^{valueOf}), (?*self*, *rdf*:*type*, *ex*:*NamedEntity*)$\star$}
("*edge*", {"*TravelsWith*"}, {"*since*"})	{(?*self*, *rdf*:*subject*, ?*source*)$\star$, (?*self*, *rdf*:*object*, ?*destination*)$\star$, (?*self*, *rdf*:*predicate*, *ex*:*TravelsWith*)$\star$, (?*self*, *ex*:*since*, "*since*"^{valueOf})$\star$}

```
1       # From _:n1
2       _:n1 rdf:type ex:Person .
3       _:n1 foaf:name "Tintin" .
4       _:n1 ex:profession "Reporter" .
5       # From _:n2
6       _:n2 foaf:name "Snowy" .
7       _:n2 rdf:type ex:NamedEntity .
8       # From _:e1
9       _:e1 rdf:subject   _:n1 .
10      _:e1 rdf:object    _:n2 .
11      _:e1 rdf:predicate _:TravelsWith .
12      _:e1 ex:since      1978 .
```

Listing 3. The RDF graph produced by the application of the well-behaved context ctx_{TTWB} on the running example PG *BTT*

5.2.3. Complexity analysis and implementation

PRSC well-behaved contexts will be proved to be reversible in Section 5.3, meaning that producing an RDF graph from them will preserve all information stored in the PG. It is therefore important to be able to determine if in practice, it is possible to compute if a PRSC context is well-behaved.

Lemma 2. *The values through κ of two given terms are either disjoint or equal:*

Proof. Consider any atomic RDF term t:

- If $t \in I$, its value through κ is the singleton composed of the element t. Other terms can not map κ to a super-set of the singleton $\{t\}$, in particular no term can be mapped to I.
- If $t \in L \cup P$, the value through κ is L. No other term can be mapped to a super-set or a subset of L.
- If $t \in B \cup pvars$, the value through κ is B. No other term can be mapped to a super-set of a subset of B.

As L, I and B are pairwise disjoint, for two given atomic RDF terms, the value through κ is either disjoint or equal.

For two given RDF triples composed of atomic terms, their value through κ are equals to the Cartesian product of the value through κ of the components. As the values through κ of their components are either equals or disjoints, the values through κ of the triples are also either equals or disjoints. By induction, this is true for any two RDF triples, even if their subject or object are also triples. □

Remark 11 (Implementing κ and complexity analysis)**.** The function κ is defined to return sets, some of them being infinite sets. While this definition is useful to prove different theorems in this paper, it is not practical from an implementation perspective.

Let λ and δ be two distinct values that are not members of the set I. We propose below an alternative function κ_{impl} to be used instead of κ in algorithms:

$$\kappa_{impl}(x) = \begin{cases} \{\kappa(triple) | triple \in x\} & \text{if } x \in RdfTriples \cup Templates \\ (\kappa_{impl}(s), \kappa_{impl}(p), \kappa_{impl}(o)) & \text{if } x = (s, p, o) \in RdfTriples \cup Templates \\ \lambda & \text{if } x \in L \cup P \\ x & \text{if } x \in I \\ \delta & \text{if } x \in B \cup pvars \end{cases}$$

Compared to Definition 22, we replaced:

- the singleton $\{x\}$ with x, in the case where $x \in I$,
- the sets L and B with two constants λ and δ that are not elements of I,
- the cross product with a simple triple of the values returned for each element of x when x is a triple.

The complexity of the $\kappa_{impl}(x)$ is:

- For any x that is not a triple nor a graph, calls to this function can be done in constant time, by simply checking the type of x.
- When x is a triple, calls to this function involves recursive calls up to the depth of x, which we consider to be bounded by a constant (see Section 4.6.2). So it is also done in constant time.
- When x is a graph, calls to this function involves calling κ_{impl} on each triple of the graph. As the call on a triple is constant, the call on the graph x has a linear complexity depending on the size of the graph.

Note that:

- For two triples, checking if their value through κ_{impl} are equals can be done in constant time.
- Thanks to Lemma 2, checking if the value through κ_{impl} of a triple t is included in the value through κ_{impl} of a graph *tps* can be done in linear time by iterating on each triple *tp* of the graph *tps* and comparing the values through κ_{impl} of the triples t and *tp*.

Remark 12 (Complexity of checking if a PRSC context is a well-behaved)**.** The first task to check if a context *ctx* is well-behaved consists in computing the value through κ of all triples used in it. As the depth of a template triple is considered to be negligible, the complexity is the number of template triples, bounded to the number of types multiplied by the size of the biggest template graph: $\mathcal{O}(NbTypes * BiggestTemplateSize)$.

After the value through κ of all template triples have been computed, for each type, we need to check if the type complies with the three criterion exposed in the Definition 24.

- The *element provenance criterion* consists in checking if ?*self* is in all templates triples of all type. This task has an $\mathcal{O}(1)$ complexity for each template triple and an overall $O(NbTypes * BiggestTemplateSize)$ complexity for the whole context.
- The *signature template triple* consists in checking if there is at least one signature template triple in the template graph of all types, *i.e.* checking if the value through κ of one of the template triples of each type is not contained in the set of the value through κ of the other types template graph. As hash sets make the membership check constant, this task has an $\mathcal{O}(NbTypes)$ complexity for a single template triple candidate, and an overall $O(NbTypes * BiggestTemplateSize * NbTypes)$ for the whole context.
- For a given type, checking the *no value loss* criterion consists in checking if a *unique* template triple can be found in the template graph for each placeholder, *i.e.* a placeholder corresponding to each property keys in the type; and if the PG element is an edge, the ?*source* and ?*destination* placeholders must also be found. Thanks to hash sets, checking if a template triple is *unique* inside its template graph is constant. Implementing the test by following the definition leads to an $\mathcal{O}(TypeComplexity * BiggestTemplateSize)$ complexity for each type and an $\mathcal{O}(NbTypes * TypeComplexity * BiggestTemplateSize)$ complexity for the whole context.

The final complexity of checking if a context is a well-behaved PRSC context is:

$$\mathcal{O}\big(NbTypes * BiggestTemplateSize * (NbTypes + TypeComplexity)\big)$$

5.3. Reversion algorithm

Algorithm 2 aims to convert an RDF graph, that was produced from a PG and a known well-behaved context, into the original PG.

It is a four steps algorithm: 1) it finds the elements of the PG, by assuming they are the same as the blank node in the RDF graph, 2) it gives a type to all PG elements with the *FindTypeOfElements* function in Algorithm 3,[10] 3) it assigns each triple to a single PG element, corresponding to the production of the *build* function, with the *AssociateTriplesWithElements* function in Algorithm 4, and 4) it looks for the source, destination and properties of all elements with the *buildpg* function in Algorithm 5.

[10] To help the comprehension of Algorithm 3, we recall that for a given set A, the mathematical notation $\exists! a \in A, somepredicate(a)$ means that in the set A, there is one and only one element, denoted by a, that matches *somepredicate*. By extension, $\exists! a \in A$ means that there is one and only one element in the set A that is denoted by a.

Algorithm 2: The main algorithm to convert back an RDF graph into a PG by using a context

Input: $rdf \subset RDFTriples$, $ctx \in Ctx^+$
Output: A BPG or error

Main Function *RDFToPG(rdf, ctx)*:
  $Elements \leftarrow BNodes(rdf)$
  $typeof \leftarrow FindTypeOfElements(rdf, ctx, Elements)$
  $builtfrom \leftarrow AssociateTriplesWithElements(rdf, Elements, typeof)$
  return $buildpg(ctx, Elements, typeof, builtfrom)$

Algorithm 3: Associate the elements of the future PG with their types

Input: $rdf \subset RDFTriples$, $ctx \in Ctx^+$, $Elements = BNodes(rdf)$
Output: A mapping between *Elements* and $Dom(ctx)$ or error

Function *FindTypeOfElements(rdf, ctx, Elements)*:
  $typeof \leftarrow \{\}$
  forall *element* $m \in Elements$ **do**
    /* Find possible types */
    $candtypes_{nodes} \leftarrow \{\}$
    $candtypes_{edges} \leftarrow \{\}$
    forall *triple* $t \in rdf | m \in t$ **do**
      forall $type \in Dom(ctx)$ **do**
        if $\kappa(sign_{ctx}(type)) = \kappa(t)$ **then**
          if $kind(type) =$ *"node"* **then**
            $candtypes_{nodes} \leftarrow candtypes_{nodes} \cup \{type\}$
          else
            $candtypes_{edges} \leftarrow candtypes_{edges} \cup \{type\}$
    /* Choose a type */
    if $(\exists! type \in candtypes_{nodes})$ *or* $(\exists! type \in candtypes_{edges}$ *and* $candtypes_{nodes} = \emptyset)$ **then**
      $typeof(m) \leftarrow type$
    else
      raise *Error(No type found)*
  return *typeof*

Further subsections prove that for all $ctx \in Ctx^+$, for all PGs pg, applying these algorithms to $rdf = prsc(pg, ctx)$ actually produces pg, meaning that the reversion algorithm is a sound and complete implementation of $prsc^{-1}$ for well-behaved contexts. Applying this algorithm to an arbitrary RDF graph and/or an arbitrary context is out of the scope of this paper.

5.3.1. Finding the elements of the PG

The first step of the algorithm relies on the assumption that the blank nodes of the RDF graph and the elements of the original PG are the same.

Theorem 2 (Equality between the elements of a PG and the blank nodes of the RDF graph)**.** *If the RDF graph rdf has been produced from a PG pg and a PRSC well-behaved context ctx, then the set of all blank nodes of rdf is the set of PG elements of pg.*

$$\forall pg \in BPGs, ctx \in Ctx^+_{pg}, rdf = prsc(pg, ctx), \quad N_{pg} \cup E_{pg} = BNodes(rdf)$$

Algorithm 4: Associate each triple to the element that has produced it

Input: $rdf \subset RDFTriples$, $Elements = BNodes(rdf)$, $typeof : Elements \mapsto Type$
Output: A mapping $Elements \to 2^{RdfTriples}$ or error

Function *AssociateTriplesWithElements*(rdf, $Elements$, $typeof$)**:**
  $builtfrom \leftarrow \{\}$
  forall $b \in Elements$ **do** $builtfrom(b) \leftarrow \{\}$
  forall $td \in rdf$ **do**
    $bns \leftarrow \{term \in td | term \in B\}$
    if $(\exists! b \in bns)$ *or* $(\exists! b \in bns | kind(typeof(b)) = \text{“edge”})$ **then**
      $builtfrom(b) \leftarrow builtfrom(b) \cup \{td\}$
    else
      `/* No blank node in bns, or multiple PG nodes but no PG edges, or multiple PG edges */`
      raise *Error(No element provenance)*
  return *builtfrom*

Proof.
- The *build* function, described in Section 4.5, produces specific triples depending on the given template. The template graphs cannot contain blank nodes: the blank node produced by *prsc* are forced to be the elements of the converted BPG. So $BNodes(rdf) \subseteq N_{pg} \cup E_{pg}$.
- From Remark 8, we know that $ctx(typeof_{pg}(m))$ contains at least one triple pattern tp. Combined with the *element provenance* criterion from Definition 24-1, we know that $?self \in tp$. When β is applied to tp, a triple that contains m is forced to appear, meaning that $N_{pg} \cup E_{pg} \subseteq BNodes(rdf)$. □

Theorem 2 proves the correctness of the $Elements \leftarrow BNodes(rdf)$ step in Algorithm 2.

5.3.2. Finding the type related to each element

In this part of the proof, we show that the *FindTypeOfElements* function from Algorithm 3 is correct, *i.e.* it is able to find back the right type of all elements m in the original pg graph.

Lemma 3. *If a data triple shares the same value through κ as one of the signature triples of a type (Definition 24-2), then the element from which the data triple was produced must be of this type:*

$$\forall td \in rdf, \forall type \in Dom(ctx), \forall m \in N_{pg} \cup E_{pg},$$
$$\left[\kappa(td) = \kappa\left(sign_{ctx}(type)\right) \wedge td \in build\left(ctx\left(typeof_{pg}(m)\right), pg, m\right)\right] \Rightarrow typeof_{pg}(m) = type$$

Proof. $\forall td \in rdf, \forall type \in Dom(ctx), \forall m \in N_{pg} \cup E_{pg}$

Assuming (A) $\kappa(td) = \kappa\left(sign_{ctx}(type)\right)$

$td \in build\left(ctx\left(typeof_{pg}(m)\right), pg, m\right)$

$\Rightarrow \exists tp \in ctx\left(typeof_{pg}(m)\right) \left| \kappa(td) = \kappa(tp)\right.$ [Lemma 1]

$\Rightarrow \exists tp \in ctx\left(typeof_{pg}(m)\right) \left| \kappa\left(sign_{ctx}(type)\right) = \kappa(tp)\right.$ [A]

$\Rightarrow \exists tp \in ctx\left(typeof_{pg}(m)\right) \left| \kappa\left(sign_{ctx}(type)\right) = \kappa(tp) \subseteq \kappa\left(ctx\left(typeof_{pg}(m)\right)\right)\right.$ [$tp \in ctx(typeof_{pg}(m))$ and by construction of κ]

$\Rightarrow typeof_{pg}(m) = type$ [Signature template triple in Definition 24-2]

□

Algorithm 5: Produce a PG from the previous analysis of the elements and triples

Input: $ctx \in Ctx^{+}$, $Elements \subset B$, $typeof : Elements \to Type$, $builtfrom : Elements \to 2^{RdfTriples}$
Output: A BPG or error

Function *buildpg*(*ctx*, *Elements*, *typeof*, *builtfrom*)**:**
  g is initialized to the empty PG
  forall $b \in Elements$ **do**
    $labels_g(b) \leftarrow labels(typeof(b))$
    if $kind(typeof(b)) =$ "*edge*" **then**
      $src_g(b) \leftarrow extract(?source, builtfrom(b), ctx(typeof(b)))$
      $dest_g(b) \leftarrow extract(?destination, builtfrom(b), ctx(typeof(b)))$
      $N_g \leftarrow N_g \cup \{src_g(b), dest_g(b)\}$
      $E_g \leftarrow E_g \cup \{b\}$
    else
      $N_g \leftarrow N_g \cup \{b\}$
    forall $key \in keys(typeof(b))$ **do**
      $properties_g(b, key) \leftarrow extract(key, builtfrom(b), ctx(typeof(b)))$
  return g

Function *extract*(*placeholder*, *tds*, *tps*)**:**
  $values \leftarrow \{\}$
  forall $tp \in tps | unique(tp, tps) \wedge placeholder \in tp$ **do**
    $samekappa \leftarrow \{td \in tds | \kappa(td) = \kappa(tp)\}$
    if $\|samekappa\| \neq 1$ **then raise** *Error(Unique data triple is not unique)*
    $td \leftarrow$ the only element in *samekappa*
    $answer \leftarrow$ The term from *td* that is at the same place as *placeholder* in *tp*
    $values \leftarrow values \cup \{answer\}$
  if $|values| \neq 1$ **then raise** *Error(Not exactly one value for a placeholder)*
  $answer \leftarrow$ The only member of *values*
  if $placeholder \in P$ **then**
    return $toLiteral^{-1}(answer)$
  else
    return *answer*

Definition 25 (Formalizing *candtypes*)**.** For a given blank node/PG element b, $candtypes_{nodes}$ and $candtypes_{edges}$, introduced in Algorithm 3, can be formally defined as:

$$candtypes_{nodes}(b) = \big\{type \in Dom(ctx) \big| kind(type) = \text{“}node\text{”} \wedge \exists td \in rdf | b \in td \wedge \kappa\big(sign_{ctx}(type)\big) = \kappa(td)\big\}$$

$$candtypes_{edges}(b) = \big\{type \in Dom(ctx) \big| kind(type) = \text{“}edge\text{”} \wedge \exists td \in rdf | b \in td \wedge \kappa\big(sign_{ctx}(type)\big) = \kappa(td)\big\}$$

They give the set of all node types and edge types, respectively, for which one of their signature triple could have produced a triple with b.

Theorem 3 (*candtypes* correctness)**.** *Even though the candtypes functions are defined by only used the used context and the produced RDF graph, they can be used to compute the type of any blank node in the original PG:*

Table 10
A simple view of Theorem 3

	$\lvert candtypes_{nodes}(b)\rvert$	$\lvert candtypes_{edges}(b)\rvert$
$b \in N_{pg}$	1	any
$b \in E_{pg}$	0	1

- $\forall b \in N_{pg}$, $candtypes_{nodes}(b) = \{typeof_{pg}(b)\}$
- $\forall b \in E_{pg}$, $candtypes_{nodes}(b) = \emptyset$ *and* $candtypes_{edges}(b) = \{typeof_{pg}(b)\}$.

Table 10 provides an overview of the cardinality of the different candtypes sets.

Proof. $\forall b \in BNodes(rdf), \forall type \in candtypes_{nodes}(b)$
Per Definition 25, $kind(type) = \text{“}node\text{”} \wedge \exists td \in rdf | b \in td \wedge \kappa(sign_{ctx}(type)) = \kappa(td)$.
We are going to restrict the portion of the graph *rdf* where such triples *td* may be located:

$$
\begin{aligned}
& td \in rdf \\
& \Leftrightarrow td \in \bigcup_{m \in N_{pg} \cup E_{pg}} build\big(ctx\big(typeof(m)\big), pg, m\big) && \text{[Definition of } rdf \text{ / } prsc\text{]} \\
& \Rightarrow td \in \bigcup_{m \in N_{pg} \cup E_{pg} | typeof(m) = type} build\big(ctx(type), pg, m\big) && \left[\begin{array}{c}\kappa(td) = \kappa(sign_{ctx}(type)) \\ \text{and Lemma 3}\end{array}\right] \\
& \Rightarrow td \in \bigcup_{m \in N_{pg} | typeof(m) = type} build\big(ctx(type), pg, m\big) && \big[kind(type) = \text{“}node\text{”}\big]
\end{aligned}
$$

- We see that all triples *td* contributing to $candtype_{nodes}(b)$ must have been produced by the signature triple template applied to a node from the PG. Also remember that *td* must contain b.
- If $b \in N_{pg}$, then the signature triple of $ctx(typeof_{pg}(b))$ must have generated a *td* containing b (since it must contain ?*self*, according to Definition 24-1), so $typeof_{pg}(b) \in candtype_{nodes}(b)$. Furthermore, no other node can produce a *td* containing b (?*self* is the only blank node placeholder in node type templates), so $candtype_{nodes}(b)$ can not contain any other type. Therefore $candtype_{nodes}(b) = \{typeof_{pg}(b)\}$.
- If $b \in E_{pg}$, it is impossible to produce the blank node b from any node $m \in N_{pg}$ (again, ?*self* is the only blank node placeholder in node type templates). No *td* containing b can be found, so $candtypes_{nodes}(b)$ is empty.

The reasoning for $candtypes_{edges}(b)$ when b is an edge is similar to the one for $candtypes_{nodes}(b)$ when b is a node: only b can produce triples containing itself, and it will, because having at least one signature triple with ?*self* is imposed by Definition 24. So $candtype_{edges} = \{typeof_{pg}(b)\}$.

Finally, a blank node $b \in N_{pg}$ can appear in any number of triples that share the same value through κ with an edge *signature template triple*: an edge *signature template triple* can contain ?*source* or ?*destination*, that can be mapped to any node depending on the PG. So $candtype_{edges}(b)$ can contain an arbitrary number of types in that case. □

Remark 13. Theorem 3 not only shows that the *FindTypeOfElements* function in Algorithm 3 will always find the right $typeof_{pg}$ function by using *candtypes*, *i.e.* that it is computable from *rdf* and *ctx*, but Table 10 also explicitly shows that the *Error(No type found)* scenario can not appear if the RDF graph was produced from a PG, making the *FindTypeOfElements* function both sound and complete.

Remark 14 (Using different signatures to determine the types)**.** As mentioned previously, the choice of the signature template triple $sign_{ctx}(type)$ of a given type is not important for the reversion algorithm. Choosing one or another signature template triple only leads to other data triples being used to determine the type of the elements. However, the end result does not change.

5.3.3. Finding the generated triples for each PG element

For each PG element, given the produced RDF graph and the type of this PG element, we are able to compute the list of RDF triples produced from this PG element. In other words, Algorithm 4 correctly partitions the RDF graph into sub-graphs describing each element of the original PG.

Theorem 4. *In Algorithm 4, assuming that the passed value of typeof is equal to* $typeof_{pg}$, $\forall m \in N_{pg} \cup E_{pg}$, $build(ctx(typeof_{pg}(m)), pg, m) = builtfrom(m)$.

Proof. As $rdf = \bigcup_{m \in N_{pg} \cup E_{pg}} build(ctx(typeof(m)), pg, m)$, each triple $td \in rdf$ is a member of at least one $build(ctx(typeof(m)), pg, m)$. For all triples $td \in build(ctx(typeof(m)), pg, m)$, the *element provenance* criterion ensures that $m \in td$. So the first step that consists in listing in the set *bns* the blank nodes in *td*, and consider that *m* is part of the set *bns* is correct: the actual element *m* is in the set.

The algorithm associates each triple *td* with a single $builtfrom(m)$:

- Let's first recall that blank nodes in *rdf* can only be produced via the placeholders from *pvars*: ?*self*, ?*source*, and ?*destination*. Let's also recall that every triple pattern in the Well-behaved context *ctx* must contain ?*self* (per the *element provenance* criterion).
- If *bns* contains only one blank node *m*, then *m* must come from ?*self*, and the corresponding triple pattern must then belong to $ctx(typeof(m))$. *td* must then have been produced by $build(ctx(typeof(m)), pg, m)$ so putting it in $builtfrom(m)$ is correct.
- If *bns* contains multiple blank nodes, *td* must have been produced by a template triples with several placeholders from *pvars*.
  - Node template graphs can contain only one placeholder from *pvars*: ?*self*. No $m \in N_{pg}$ could then have produced *td*. It follows that *td* must have been produced by the template graph of an edge.
  - Edge template graphs can contain several placeholders from *pvars*. But by definition of β, only ?*self* can be mapped to an edge (when $m \in E_{pg}$); ?*source* and ?*destination* are always mapped to nodes. Of the multiple blank nodes in *bns*, exactly one of them, *m*, must therefore be an edge, and come from ?*self*. Following the same reasoning as above, when *bns* contained only one blank node, we conclude that *td* must then have been produced by $build(ctx(typeof(m)), pg, m)$ and that putting it in $builtfrom(m)$ is correct.
- *Error(No element provenance)* will never be raised if *rdf* was produced by *prsc*: each triple will contain at least one blank node generated from ?*self* (per the *element provenance* criterion), and if there are multiple blank nodes we showed that there must be only one edge blank node.

As each triple in *rdf* is attributed in $builtfrom(m)$ to the right element *m* that produced it from $build(ctx(typeof_{pg}(m)), pg, m)$, $\forall m \in N_{pg} \cup E_{pg}$, $builtfrom(m) = build(ctx(typeof_{pg}(m)), pg, m)$. □

5.3.4. Building the PG element

Projecting Property Graphs As an RDF graph is defined as a set of RDF triples, any subset of that set, as well as the union of two RDF graphs, are formally defined and are also RDF graphs. Algorithm 5 constructs back the original PG in an iterative manner. To prove its correctness, we need operators similar to $\subset$ and $\cup$ for RDF graphs, but for our formalization of PGs.

In this section, the projection of a Property Graph is defined by focusing only on a single **PG element**, node or edge. The concept of merging PGs, which is the inverse of the projection, is also defined.

Let *pg* be a PG.

Definition 26 (π projection of a Property Graph on an element)**.** The π projection of a PG on a node is equal to the PG with only the node itself. The π projection of a PG on an edge is the edge, and its source and destination nodes without the labels and properties of these nodes.

$\forall m \in N_{pg} \cup E_{pg}$, $\pi_m(pg)$ is a PG such as:

- If $m \in N_{pg}$, $N_{\pi_m(pg)} = \{m\}$, $E_{\pi_m(pg)} = \emptyset$, $src_{\pi_m(pg)} = dest_{\pi_m(pg)} = \emptyset \rightarrow \emptyset$
- If $m \in E_{pg}$, $N_{\pi_m(pg)} = \{src_{pg}(m), dest_{pg}(m)\}$, $E_{\pi_m(pg)} = \{m\}$, $src_{\pi_m(pg)} = \{m \mapsto src_{pg}(m)\}$, $dest_{\pi_m(pg)} = \{m \mapsto dest_{pg}(m)\}$

– $\forall x \in N_{\pi_m(pg)} \cup E_{\pi_m(pg)}$,

$$labels_{\pi_m(pg)}(x) = \begin{cases} labels_{pg}(x) & \text{if } x = m \\ \emptyset & \text{otherwise} \end{cases}$$

– $\forall key \in keys_{pg}(m), properties_{\pi_m(pg)}(m, key) = properties_{pg}(m, key)$, all other values are undefined.

Definition 27 (Property Graph merge operator $\oplus$)**.** The merge operator $\oplus$ is the inverse of the projection operator π. It can only be used on two PGs that are compatible, *i.e.* (1) a PG element defined as a node is not defined as an edge in the other, (2) an edge defined in both PGs have the same source and destination in both, and (3) if in both PGs, the value of a property key on the same PG element is defined, the values should be the same. The $\oplus$ operator builds a PG with the PG elements, labels and properties of both PGs.

We now define the $\oplus$ merge operator on property graphs. $\forall (pg', pg'') \in PGs^2, \oplus(pg', pg'')$ (or $pg' \oplus pg''$) is defined only if:

– $E_{pg'} \cap N_{pg''} = \emptyset \wedge N_{pg'} \cap E_{pg''} = \emptyset$
– $src_{pg'}$ is compatible with $src_{pg''}$ and $dest_{pg'}$ is compatible with $dest_{pg''}$ (see compatibility definition in Section 3.2).
– $properties_{pg'}$ is compatible with $properties_{pg''}$.

Its value is $\oplus(pg', pg'') = pg$ with:

– $N_{pg} = N_{pg'} \cup N_{pg''}$
– $E_{pg} = E_{pg'} \cup E_{pg''}$
– $src_{pg} : E_{pg} \to N_{pg}, src_{pg} = src_{pg'} \cup src_{pg''}$.
– $dest_{pg} : E_{pg} \to N_{pg}, dest_{pg} = dest_{pg'} \cup dest_{pg''}$.
– $\forall m \in N_{pg} \cup E_{pg}$,

$$labels_{pg}(m) = \begin{cases} labels_{pg'}(m) \cup labels_{pg''}(m) & \text{if both are defined} \\ labels_{pg'}(m) & \text{if only } labels_{pg'}(m) \text{ is defined} \\ labels_{pg''}(m) & \text{if only } labels_{pg''}(m) \text{ is defined} \end{cases}$$

– $properties_{pg} : (N_{pg} \cup E_{pg}) \times Str \to V, properties_{pg} = properties_{pg'} \cup properties_{pg''}$.

Lemma 4. *$\oplus$ is commutative, associative, and the neutral element is the empty PG $pg_\emptyset$*

Proof. (Sketch). $\oplus$ is defined by using the $\cup$ operator, which is commutative, associative and whose neutral element is $\emptyset$. The equivalent of $\emptyset$ for PGs is $pg_\emptyset$. □

Theorem 5. *The $\oplus$ merge of the π projection of a PG on all its PG elements is equal to the PG itself:*

$$\forall pg \in PGs, \quad pg = \bigoplus_{m \in N_{pg} \cup E_{pg}} \pi_m(pg)$$

Proof. The proof is provided in Appendix A. □

While the $\oplus$ operator has been primarily designed as the inverse operation of the projection operator π, it can only merge two PG that are consistent between themselves. If a PG defines an entity as a node, the other PG can not define it as an edge. If a PG has a given source and destination for a given edge, the other one can not define another source or destination. Properties must also be consistent in both PGs. The presented version of the merge operator can only add information, and in the case merging two PGs would lead to an inconsistent PG, the merge operator is undefined.

Relationship between the prsc function and projections We are now going to redefine the *prsc* function using the π operator.

The RDF graph built by *prsc* from a PG pg with a context ctx is equal to:

$$rdf = \bigcup_{m \in N_{pg} \cup E_{pg}} build\big(ctx\big(typeof_{pg}(m)\big), pg, m\big)$$

The *build* function is defined in such a way that the RDF triples it produces from an element m are only influenced by:

- m itself.
- Its labels, *i.e.* $labels_{pg}(m)$.
- Its property values, *i.e.* $\forall key, properties_{pg}(m, key)$.
- If m is an edge, its source and destination nodes, *i.e.* $src_{pg}(m)$ and $dest_{pg}(m)$.
- The template graph $ctx(typeof_{pg}(m))$.

Therefore the following equality can be asserted, $\forall pg \in BPGs, \forall m \in N_{pg} \cup E_{pg}, \forall ctx \in Ctx_{pg}$:

$$build\big(ctx\big(typeof_{pg}(m)\big), \boldsymbol{pg}, m\big) = build\big(ctx\big(typeof_{pg}(m)\big), \boldsymbol{\pi_m(pg)}, m\big)$$

$\pi_m(pg)$ can be considered as the minimal required Property Graph to produce the RDF triples related to the element m in the PG pg. If we can prove that the reversion algorithm constructs all $\pi_m(pg)$ graphs and merges them with the $\oplus$ operator, then it means that we have properly reconstructed the pg PG.

Completing the proof of the reversion algorithm We are now back to proving that for all well-behaved contexts ctx, the *RDFToPG* function presented in Algorithm 2 is an implementation of the $prsc^{-1}$ function. pg is a PG for which we know the value of $prsc(pg, ctx) = rdf$. We are focusing on the last line of Algorithm 2, where the *buildpg* function is invoked.

To prove the correctness of the *buildpg* function in Algorithm 5, starting from an empty PG g, we are going to show that at each iteration, we are adding to the PG g the π projection of pg on an element m. After iterating on all elements, as we merged the π projection of all PG elements, the PG g ends up being equal to the PG pg itself.

Lemma 5 (Merging the projection of one PG element to the reconstructed PG)**.** *In Algorithm 5, assuming that the typeof parameter is equal to* $typeof_{pg}$ *and buildfrom is a total function that maps all PG elements* b *to* $build(ctx(typeof_{pg}(b)), pg, b)$*, at the end of an iteration of an element* $b \in N_{pg} \cup E_{pg}$ *after line 13, the computed PG* g_{after} *is equal to* $g_{before} \oplus \pi_b(pg)$*, where* g_{before} *is the PG* g *at the beginning of the iteration between lines 3 and 4.*

Proof. The PG $\pi_b(pg)$ is described in Table 11. Bold values are the ones for which we need to prove that we compute the correct value: $src_g(b)$, $dest_g(b)$ and $properties_g(b, key)$. Other values are trivially correct by construction.

In the following, we want to check that $extract(?source, build(\pi_b(pg), pg, b), ctx(typeof(b)))$ properly returns $src_{\pi_b(pg)}$. Proofs for $?destination$ / $dest_{\pi_b(pg)}$ and $key \in keys_{typeof_{pg}(b)}$ / $properties_{\pi_b(pg)}(b, key)$ are identical.

Table 11
Description of the PG projection that is built in Algorithm 5

	$b \in N_{pg}$	$b \in E_{pg}$
$N_{\pi_b(pg))}$	$\{b\}$	$Img(src_{\pi_b(pg)}) \cup Img(dest_{\pi_b(pg)})$
$E_{\pi_b(pg)}$	$\emptyset$	$\{b\}$
$src_{\pi_b(pg)}$	$\emptyset \to \emptyset$	$b \mapsto \boldsymbol{src_g(b)}$
$dest_{\pi_b(pg)}$	$\emptyset \to \emptyset$	$b \mapsto \boldsymbol{dest_g(b)}$

	$b \in N_{pg} \cup E_{pg}$
$labels_{\pi_b(pg)}$	$\{b \mapsto labels(type)\}$
$properties_{\pi_b(pg)}$	$\bigcup_{key \in keys(type)} \{(b, key) \mapsto \boldsymbol{properties_g(b, key)}\}$

The *values* set is filled by iterating on all *tp* such that $unique(tp, tps) \wedge ?source \in tp$. The *no value loss* criterion ensures that at least one such template triple exists, so the loop in *extract* is iterated at least once.

Theorem 1 ensures that the built set *samekappa* in the loop of the *extract* function will always have 1 element, that we name *td*. *Error(Unique data triple is not unique)* may never be raised if *rdf* was produced by PRSC. By definition of the *build* function, ?*source* in *tp* and $src_{\pi_b(pg)}$ in *td* are at the same position.

After the loop, because only $src_{\pi_b(pg)}$ is added to *values* in the loop, *Error(Not exactly one value for a placeholder)* may never be raised.

The last instructions differ for ?*source* / ?*destination* and P. In the case of ?*source* and ?*destination*, the obtained value is directly the value of the PG node; in the case of P, the obtained RDF literal needs to be converted into the proper PG property value, which is possible because $toLiteral^{-1}$ is assumed to be computable in Section 4.5.

extract properly computes the values that are missing in $\pi_b(pg)$. When these values are extracted, they are directly merged with the $\cup$ operator into the g property graph. Values that were already known or can be computed from the values that were just extracted, *i.e.* $labels_{\pi_m(pg)}$, $N_{\pi_m(pg)}$ and $E_{\pi_m(pg)}$, are also merged into g.

As all values of $\pi_b(pg)$ are merged into g_{before}, $g_{after} = g_{before} \oplus \pi_m(pg)$ □

Remark 15 (Completeness of *buildpg*)**.** In the case where *rdf* is built from a PG *pg*, the value that a placeholder is mapped to is the same everywhere, so we never run at the risk of encountering multiples values, *i.e.* Error(Not exactly one value for a placeholder) is never raised. Furthermore, the proof of Lemma 5 shows that *Error(Unique data triple is not unique)* may not be raised, because we know that each unique template triple has produced one data triple.

Theorem 6 (Merging the projection of all PG elements to the reconstructed PG)**.** *Under the same assumptions as Lemma 5, the PG returned by Algorithm 5 is the original pg, the PG that was used to produce the RDF graph* $rdf = prsc(pg, ctx)$.

Proof. The PG g in the algorithm is initialized to $pg_\emptyset$. Lemma 5 shows that after each iteration in the loop with an element b, the PG g is $\oplus$-merged with the PG $\pi_b(pg)$. The loop iterates on all elements in the PG *pg*, so after all the iterations, the PG g is equal to:

$$
\begin{aligned}
g &= pg_\emptyset \oplus \bigoplus_{b \in N_{pg} \cup E_{pg}} \pi_b(pg) \\
&= \bigoplus_{b \in N_{pg} \cup E_{pg}} \pi_b(pg) && [pg_\emptyset \text{ is the neutral element of } \oplus] \\
&= pg && [\text{Theorem 5}] \quad \square
\end{aligned}
$$

As *buildpg* in Algorithm 5 correctly reconstructs *pg*, and as its value is directly returned by the *RDFToPG* function in Algorithm 2, we have finally proven that the latter is a sound and complete implementation of the $prsc^{-1}$ function for any well-behaved PRSC context *ctx*.

5.3.5. Complexity analysis

Let us now discuss the complexity of the *RDFToPG* function described in Algorithm 2. In this discussion, a new metric is considered: the number of triples in the RDF graph: $NbTriples = |rdf|$. It is assumed that we have first checked if the context *ctx* is a well-behaved PRSC context, and computed the $sign_{ctx}$ function so it can now be called in constant time.

Extracting the list of blank nodes of an RDF graph on line 2 has a linear complexity of $\mathcal{O}(NbTriples)$.

The *FindTypeOfElements* function in Algorithm 3 uses three nested loops and only uses constant time operations: its complexity is $\mathcal{O}(NbOfPGElements * NbTriples * NbTypes)$.

Trivially, Algorithm 4 has a complexity of $\mathcal{O}(NbOfPGElements + NbTriples)$.

In Algorithm 5:

- Calls to $extract(placeholder, tds, tps)$ have a complexity of $\mathcal{O}(|tps|^2 * |tds|)$:
  - It loops all triples in the template graph *tps* such that they are *unique*. Evaluating $unique(tp, tps)$ itself has an $\mathcal{O}(|tps|)$ complexity so the overall complexity of evaluating all $tp \in tps | unique(tp, tps)$ is $\mathcal{O}(|tps|^2)$.
  - Inside the loop, building the *samekappa* set forces to loop on all *tds*, multiplying the complexity by a $|tds|$ factor.

  In the context of the *buildpg* function, the *extract* function is always called with a template graph *tps* from the PRSC context and a sub-graph of the RDF graph *rdf* as *tds*. the complexity of the calls of the *extract* function in the *buildpg* function is $\mathcal{O}(BiggestTemplateSize^2 + NbTriples)$.[11]
- The *buildpg* function loops on all PG elements.
  - Notice that while $ctx(typeof(b))$ is called multiple times, it can be called once and then its value can be cached. Its cost is $\mathcal{O}(TypeComplexity * \ln(TypeComplexity))$ as mentioned in Section 4.6.3.
  - There are at most $2 + TypeComplexity$ calls of the *extract* function. All of them have a $\mathcal{O}(BiggestTemplateSize * NbTriples)$ complexity.
  - The complexity of each iteration is $\mathcal{O}(TypeComplexity * \ln(TypeComplexity) + TypeComplexity * BiggestTemplateSize^2 * NbTriples))$
- The overall complexity of the *buildpg* function is $\mathcal{O}(NbOfPGElements * TypeComplexity * (\ln(TypeComplexity) + BiggestTemplateSize^2 * NbTriples))$

The overall complexity of the *RDFToPG* function presented in Algorithm 2 presented in this section for an RDF graph produced from a PG and a well-behaved PRSC context is:

$$
\begin{aligned}
&\mathcal{O}\big(NbTriples \\
&\quad + NbTypes * NbTriples * BiggestTemplateSize \\
&\quad + NbOfPGElements + NbTriples \\
&\quad + NbOfPGElements * TypeComplexity * \big(\ln(TypeComplexity) + BiggestTemplateSize^2 * NbTriples\big)\big) \\
&= \mathcal{O}\big(NbTypes * NbTriples * BiggestTemplateSize \\
&\quad + NbOfPGElements * TypeComplexity * \big(\ln(TypeComplexity) + BiggestTemplateSize^2 * NbTriples\big)\big)
\end{aligned}
$$

The *RDFToPG* function is computable in polynomial time w.r.t. all the considered metrics, and is therefore considered as tractable. Furthermore, in our implementation, the algorithm has been optimized to significantly lower the complexity. However, for the sake of concision, we do not describe these optimizations in this paper.

5.4. Edge-unique extension

In many cases, there is only one edge of certain types between two nodes, like the "TravelWith" edge in our running example or for relationships like knowing someone, a parental relationship... For this type of edges, it is more intuitive to represent them with a simple RDF triple, and get rid of the blank node corresponding to the edge. However, Well-Behaved PRSC contexts require ?*self* in edge templates. In this section, we propose an extension to allow ?*self* to be missing in edge templates and still produce reversible conversions.

Consider the Tintin PG exposed in Fig. 1 and the context exposed in Table 12, which uses RDF-star to convert the "since" property. The output of PRSC from those two inputs is exposed in Listing 4. By looking at the produced RDF

[11] Note that if *rdf* has been generated by the *prsc* function, the number of triples in the graph *tds* is inferior or equal to the number of triples in the template graph *tps* as *tds* has been generated from *tps*. In this case, the complexity of calling *extract* in the context of the *buildpg* function is $\mathcal{O}(BiggestTemplateSize^3)$.

Table 12
A context for the Tintin PG with the "since" property

type	*ctx*(*type*)
("*node*", {"*Person*"}, {"*name*", "*job*"})	{(?*self*, *rdf*:*type*, *ex*:*Person*), (?*self*, *foaf*:*name*, "*name*"^{valueOf}), (?*self*, *ex*:*profession*, "*job*"^{valueOf})}
("*node*", ∅, {"*name*"})	{(?*self*, *foaf*:*name*, "*name*"^{valueOf})}
("*edge*", {"*TravelsWith*"}, {"*since*"})	{(?*source*, *ex*:*isTeammateOf*, ?*destination*), ((?*source*, *ex*:*isTeammateOf*, ?*destination*), *ex*:*since*, "*since*"^{valueOf})}

```
1   % Tintin node
2   _:n1 rdf:type ex:Person .
3   _:n1 foaf:name "Tintin" .
4   _:n1 ex:profession "Reporter" .
5   % Snowy node
6   _:n2 foaf:name "Snowy" .
7   % TravelsWith edge
8   _:n1 ex:isTeammateOf _:n2 .
9   << _:n1 ex:isTeammateOf _:n2 >> ex:since 1978 .
```

Listing 4. The output of PRSC for the Tintin PG and the context exposed in Table 12

graph, it appears that the RDF graph captures all the information of the PG. More generally, RDF graphs produced by this context would always be reversible as long as the source PG does not contain multiple "TravelsWith" edges between two given nodes.

Definition 28 (Edge-unique extension)**.** a) In a context *ctx*, an **edge-unique type** *edgeunq* is an edge type such that:

- *ctx*(*edgeunq*) complies with the *no value loss* criterion (as per Definition 24-3) and is not empty.
- For all template triples $tp \in ctx(edgeunq)$:
  - ?*source* $\in tp$ and ?*destination* $\in tp$
  - *tp* is a *signature template triple* (as per Definition 24-2), *i.e.* no other type has a template triple that shares its value through κ.
  - *tp* is a *unique* template triple, *i.e.* no other template triple in *ctx*(*edgeunq*) shares its value through κ.

b) A PG *pg* is said edge-unique valid for a context *ctx* if for all edge-unique types in the context, there is at most one edge of this type between two given nodes:

$$\forall e \in E_{pg}, typeof_{pg}(e) \text{ is an edge-unique type} \Rightarrow \left(\forall e' \in E_{pg}, \begin{bmatrix} typeof_{pg}(e) = typeof_{pg}(e') \\ \wedge \quad src_{pg}(e) = src_{pg}(e') \\ \wedge \quad dest_{pg}(e) = dest_{pg}(e') \end{bmatrix} \Rightarrow e = e' \right)$$

c) The *prscEdgeUnique* function is introduced to serve as a proxy to the *prsc* function to be applied only if the given PG is edge-unique valid relatively to the given context:

$$prscEdgeUnique(pg, ctx) = \begin{cases} prsc(pg, ctx) & \text{if } pg \text{ is edge-unique valid for } ctx \\ \text{undefined} & \text{otherwise} \end{cases}$$

Theorem 7 shows that *prscEdgeUnique* is reversible up to an isomorphism.

Theorem 7. *Let ctx be a context such that each type either a) matches the constraints of a type in a well-behaved PRSC context in Definition 24 or b) is an edge-unique type.*

- *For every two BPGs, pg' and pg'', such that $prscEdgeUnique(pg', ctx) = prscEdgeUnique(pg'', ctx)$, pg' and pg'' are isomorphic.*

– *There is an algorithm such that for all BPGs pg, from the RDF graph prscEdgeUnique*(pg, ctx) *and the context ctx, the algorithm computes a PG* pg' *such that prscEdgeUnique*$(pg, ctx) = prscEdgeUnique(pg', ctx)$*, i.e. from the produced RDF graph and the context, it is possible to compute a PG that is isomorphic to the original one.*

Proof. (Sketch). The context *ctx* is composed of two parts: a) the well-behaved part and b) the edge-unique part. The well-behaved part has been proved to be reversible. As template triples used for edge-unique types are signatures, their value trough κ is different from the triples produced from the value through κ of the triples of the well-behaved part: triples produced from edge-unique types are distinguishable from the rest of the RDF graph.

Denote W the set of all types in the well-behaved part and U the types in the edge-unique part. Let pg be a PG such that $rdf = prscEdgeUnique(pg, ctx)$ exists. It is possible to split pg using W and U:

$$pg = \underbrace{\bigoplus_{m \in N_{pg} \cup E_{pg} | typeof_{pg}(m) \in W} \pi_m(pg)}_{pg_W} \oplus \underbrace{\bigoplus_{u \in E_{pg} | typeof_{pg}(u) \in U} \pi_u(pg)}_{pg_U}$$

It is also possible to split *rdf* by defining an *isWellBehaved* predicate that uses κ to filter triples that come from types in the well-behaved part: $\forall td \in RdfTriples, isWellBehaved(td) \Leftrightarrow \exists type \in W, \exists tp \in ctx(type), \kappa(td) = \kappa(tp)$.

$$rdf = \underbrace{\{td \in rdf | isWellBehaved(td)\}}_{rdf_W} \cup \underbrace{\{td \in rdf | \neg isWellBehaved(td)\}}_{rdf_U}$$

From all the theorems on well-behaved contexts, there is a bijection between pg_W and rdf_W.

All template triples used in the template graph of edge-unique types are both signature and unique: from any triple in rdf_U, it is possible to find which template triple produced it. Consider an arbitrary edge u, whose type is an edge-unique type, *i.e.* $typeof_{pg}(u) \in U$. As edge-unique template graphs must also comply with the *no value loss* criterion, all properties, the source node and the destination node of u can be found in a non-ambiguous manner in rdf_U. The only missing information is the edge identity, *i.e.* the blank node u itself.

By using a fresh blank node for u, it is possible to build a PG isomorphic to $\pi_u(pg)$ from rdf_U, by extension, a PG isomorphic to pg_U from rdf_U, and by extension a PG isomorphic to pg from rdf. □

5.5. Discussion about the constraints on well-behaved PRSC contexts

In this section, we discuss the acceptability of the different constraints posed by PRSC well-behaved contexts in terms of usability. In other words, to what extent do they limit what can be achieved with PRSC?

The *no value loss* criterion on well-behaved contexts ensures that the data are still present and can be found unambiguously: as its name implies, this constraint is obviously required to avoid information loss. Therefore, it should not be perceived as overly constraining when building PRSC contexts.

The *signature template triple* is a method to force the user to type the resources, which is usually considered to be good practice. The type can either be explicit, through a triple with *rdf*:*type* as the predicate, or implicit through a property that is only used by this type. For example, the template graph for a type *Person* could contain a template triple for the form $(?self, :personId, \text{“}pid\text{”}^{valueOf})$. The constraint of a signature composed of only one triple can be considered too strong: one may want to write a context that works for all PGs. For example, many authors [20,32] propose to map each label to an RDF type or a literal used as the object of a specific predicate like `pgo:label`. More generally, users may want to use a composite key to sign their types. For these kinds of mappings, our approach of identifying the type by finding a single signature template is not sufficient. It requires finding all the signature template triples and deciding to which type they are associated, for example through a Formal Concept Analysis process. This could be studied as a future extension of the PRSC reversion algorithm.

The *element provenance* constraint may hinder the integration of RDF data coming from a PG with regular RDF data: it forces the user to keep the structure exposed in the PG, with blank nodes representing the underlying structure of the PG. The edge-unique extension enables to leverage this constraint, by avoiding representing PG edges as RDF nodes.

6. Related works

Many works already exist to address the interoperability between PGs and RDF.

A common pivot for PGs and RDF To achieve interoperability, some authors propose to store the data into another data model, and then expose the data through usual PG and RDF APIs. Angles et al. propose multilayered graphs [4], for which the OneGraph vision from Lassila et al. [23] is a more concrete version. These works propose to describe the data with a list of edges, with the source of the edge, a label and the destination of the edge. All edges are associated with an identifier, that can be used as the source or the destination of other edges. However, authors note that several challenges are raised about the way to implement the interoperability between the OneGraph model and the existing PG and RDF APIs.

In a Unified Relational Storage Scheme [35], Zhang et al. propose to store the data in relational databases. While they specify how to store both models in a similar relational database structure, they do not mention how they align the data that come from one model with the data that come from another, for example to match the PG label "Person" with the RDF type `foaf:Person`.

The Singleton Property Graph model proposed by Nguyen et al. [25] is an abstract graph model that uses the RDF Singleton Property pattern that can be implemented both with a PG and an RDF graph. They also describe how to convert a regular RDF graph or a regular PG into a Singleton Property Graph. But the use of the Singleton Property pattern induces the creation of many different predicates, which hinders the performance of many RDF database systems as shown by Orlandi et al. [26].

From PGs to RDF In terms of PG to RDF conversion, the most impactful work is probably RDF-star [17–20], an extension of the RDF model originally proposed by Olaf Hartig and Bryan Thompson to bridge the gap between PGs and RDF by allowing the use of triples in the composition of other triples. Indeed, the most blatant difficulty when converting PG to RDF is converting the edge properties. However, most PG engines support multi-edges, *i.e.* two edges of the same type between the two same nodes. On the other hand, the naive approach consisting in using the source node, the type of the edge and the destination node as respectively the subject, the predicate and the object of an RDF triple would collapse the multi-edges. Converting each edge property to an RDF-star triple that uses the former triple as its subject would lead to the properties of each multi-edge to be merged. Khayatbashi et al. [21] study on a larger scale the different mappings described by Hartig and benchmark them. By allowing triples to be used as the subject and the object of other triples, it is possible to emulate the edge properties of PGs. While these mappings allow some kind of user customization, by letting them choosing the used IRIs, they never consider using different model structures for different PG types during the same conversion. For example, consider a PG with two types of edges: one edge type with the "knows" label and one with the "marriedTo" label. The mappings described in this paper do not enable the user to model edges with the "knows" label as a predicate and edges with the "marriedTo" label as a proper RDF resource. To tackle the edge property problem, Das et al. study how to use already existing reification techniques to represent properties [12]: the modelings that do not rely on quads can be used when writing a PRSC context.

Tomaszuk et al. propose the Property Graph Ontology (PGO) [32], an ontology to describe PGs in RDF. As this solution only describes the structure of the PG in RDF, the produced data is forced to use this ontology, with the exception of other already existing RDF ontologies without further transformations. Thanks to the Neosemantics[12] plugin developed by Barrasa, Neo4j is able to benefit from RDF related tools like ontologies, and performs a 2-way conversion from and to RDF-star data. However, the PG to RDF conversion performed by Barrasa tends to affirm all triples it can, even for PG edges that may describe facts with a probability or that are time restricted: if the marriage between Alice and Bob has ended in 2017, the triple `:Alice :marriedto :Bob` should probably not be produced.

Gremlinator [31] allows users to query a PG and an RDF database by using the SPARQL language. This is a first step towards federated queries. However, it supposes that data stored in the PG and data stored in the RDF graph have a similar modeling, and it does not support RDF-star.

[12] https://github.com/neo4j-labs/neosemantics

Instead of having a fixed mapping, our work on PREC [10] propose a mapping language named PREC to drive the conversion from PG to RDF. Delva propose RML-star [13], an extension of RML [14] and R2RML [30] that introduces new RML directives to generate RDF-star triples. As discussed in Section 2, the format in which the template triples are described in this work is closer to the produced triples, at the cost of reducing the ability to produce templated IRIs or terms.

To leverage the requirement for users to manually write the mapping, Fathy et al. proposed ProGoMap [15], an engine that first generates a putative ontology for the terms in a PG, aligns this ontology with a real world ontology, and finally converts the PG to an RDF graph with an RML mapping generated from the alignment. The authors motivate the choice of automating the ontology alignment process by mentioning that writing mapping manually can be time-consuming. While this may be true, we do not think that this hinders PRSC usefulness as 1) the schema part of a PRSC context may be generated automatically, and nothing prevents a tool from generating the template triples automatically, 2) a PG schema for which writing template triples is time-consuming may be an indicator that the schema is too complex, and therefore that the structure of the data stored in the PG should be cleaned, not only to make the conversion to RDF easier, but also as a benefit for the PG itself, and 3) while the process of aligning terms of the putative ontologies to terms of real ontologies indeed increase data interoperability, it still relies on the idea that the structure of the PG is somewhat close to the structure of the desired RDF graph. However, in this paper, we advocate that it may not be the case, and that the desired method to model each edge type depends on the semantics of held by each edge type.

In general, to the best of our knowledge, most existing works only tackle the syntactic aspect of converting PG to RDF data. While this level of interoperability is appreciated, it is not enough to be able to properly use the converted data in any existing RDF application without further modification of the RDF graph. Other existing works provide a level of semantic interoperability. However, they tend to choose one of the many syntactic representations of edges that exist, despite RDF ontologies having a great variety of patterns to model knowledge. PRSC gives full control to the user, by letting them choosing both the syntactic representation and use shared vocabularies in the produced RDF graph. The use of PG schemas in PRSC contexts guides users in the process of writing the context. The study on PRSC well-behaved contexts, and the related discussion on their constraints in Section 5.5, provide information on if the PRSC context written by the user leads to a reversible conversion or not, and by consequence, if any information is lost in the process of converting the data stored in the PG into RDF.

From RDF to PGs Abuoda et al. [1] study the different RDF-star to PG approaches and identified two classes: the RDF-topology preserving transformation which converts each term into a PG node, and the PG transformation that converts literals into property values. They also evaluate the performance of these different approaches. The PRSC reversion algorithm, and the general philosophy of this work clearly falls under the latter category. The former can be considered as using a PG database to store RDF data.

Angles et al. [5] discuss different methods to transform an RDF graph into a PG. They propose different mappings, including an RDF-topology preserving one and a PG transformation. Atemezing and Hyunh [6] propose to use a mapping similar to the former to publish and explore RDF data with PG tools, namely Neo4j. However, these works offer little customization for the user.

With G2GML [11], Chiba et al. propose to convert RDF data by using queries: the output of the query is transformed into a PG by describing a template PG, similar to a Cypher insert query. This approach can be considered to be a counterpart of PRSC, but to convert RDF into PG.

PG schemas Finally, the "Property Graph needs a Schema" Working Group propose a formal definition of PG schemas [3]. Some PG engines, like TigerGraph, are based on the use of schemas. For PG engines that do not enforce a schema at creation, like Neo4j or Amazon Neptune, the schema may be extracted from the data, as proposed by Bonifati et al. [8] or Beereen [7]. As PRSC uses schemas for mapping between PGs and RDF graphs, these approaches may be used to automatically list the types existing in the PG to convert *i.e.* the target of the rule part in Listing 2. Then the user would only have to provide the way to convert these types into RDF, *i.e.* the template graph part.

7. Conclusion

This work improves interoperability between the two worlds of Property Graphs and RDF graphs. We have presented PRSC, a mapping language to convert PGs into RDF graphs. A mapping, named PRSC context, is written by the user and is driven by a schema: PG elements are converted according to their type. By letting the user configure the conversion, we aim to better integrate PG data into already existing RDF graphs: the produced RDF graphs can be made to use a specific vocabulary, or comply with specific shapes.

We have also proved that some PRSC contexts, named well-behaved PRSC contexts, are reversible: they do not induce any information loss, and therefore it is possible to reverse back to the original PG from the produced RDF graph. Finally, we broaden the realm of reversible contexts with the edge-unique extension. Other existing works focus either on describing a syntactic construction, or providing a semantic construction that relies on a specific syntactic pattern. PRSC lets the user specify the semantics, and proved the reversibility for the two most popular methods to model edges: as an RDF resource in PRSC well-behaved contexts, and as a predicate through the edge-unique extension.

For big PGs, fully converting them into RDF may not scale. For this reason, future works include studying how to use PRSC context not only for PG conversion but also to convert SPARQL queries into the usual PG query languages Cypher and Gremlin. This would not only address the scalability issues, but also avoid data duplication and help for federated queries.

The expressiveness of PRSC contexts could also be extended. As it is currently presented, PRSC contexts are unable to reproduce RDF graphs complying with some ontologies, for example the PG ontology [32]. To solve this issue, PRSC contexts should be able to introduce new blank nodes, and not be limited to the ones in the original PG. This would lead to new challenges, as the presented reversion algorithm relies on the fact that all blank nodes are PG elements.

Other extensions on expressiveness may also be interesting. For examples, types in PRSC contexts are *closed*, in the sense that a complying element must have exactly the properties of the type, barring any other. Allowing extra properties in elements of the PGs would be useful, but raises the challenge of converting properties that are not known in advance.

To let PG data further benefit from the tools that have been developed around RDF, PRSC could also be explored in two directions. The use of blank nodes for the PG elements may not be suitable in all cases, especially in a linked data context. PRSC could be extended to mint IRIs for nodes and edges of the PG, but this would require to extend the mapping language to be able to express these template IRIs. It would also require an adaptation of the reversion algorithm to be able to differentiate the minted IRIs from the "static" IRIs of the template, which would require additional precautions on well-behaved contexts.

The provided reversion algorithm does not only work for RDF graphs that were produced by PRSC, but can work on any compatible RDF graph. One way to use it would be to use PRSC to convert a PG to an RDF graph, modify the produced RDF graph with RDF-specific tools, *e.g.* by using a reasoner designed for RDF graphs, and then transform back the RDF graph into a PG, which could be considered as equivalent as using a reasoner on a PG. However, this requires to formally characterize the modifications that can be performed on the RDF triples while maintaining the ability to convert it back to a PG.

Appendix A. Proof of projecting and merging back a Property Graph

In this section, we expose the proof for Theorem 5

A.1. Extra mathematical elements

Definition 29 (Restriction)**.** For all functions f, for all sets X, $f|_X = \{(x, f(x))|x \in X \cap \mathrm{Dom}(f)\}$. $f|_X$ is called the restriction of the function f to the set X. In other words, the restriction of a function by a set X is equal to a function in which the domain is restricted to the elements of the set X.

Remark 16. The restriction of a function to its domain is equal to the function itself: $f|_{\mathrm{Dom}(f)} = \{(x, f(x))|x \in \mathrm{Dom}(f)\} = f$.

Remark 17. A functional definition of Definition 29 would be, for all functions f, $f|_X : x \mapsto f(x)$ if $x \in X \cap \mathrm{Dom}(f)$, undefined otherwise.

Lemma 6. *For all functions f, for all sets X_1 and X_2, the union of the function restricted by the two sets is equal to the function restricted by the union of the two sets: $f|_{X_1} \cup f|_{X_2} = f|_{X_1 \cup X_2}$.*

Proof.

$$\begin{aligned}
&f|_{X_1} \cup f|_{X_2} \\
&\quad = \big\{\big(x, f(x)\big)|x \in X_1 \cap \mathrm{Dom}(f)\big\} \cup \big\{\big(x, f(x)\big)|x \in X_2 \cap \mathrm{Dom}(f)\big\} \\
&\quad = \big\{\big(x, f(x)\big)|x \in \big(X_1 \cap \mathrm{Dom}(f)\big) \cup \big(X_2 \cap \mathrm{Dom}(f)\big)\big\} \\
&\quad = \big\{\big(x, f(x)\big)|x \in (X_1 \cup X_2) \cap \mathrm{Dom}(f)\big\} \\
&\quad = f|_{X_1 \cup X_2}
\end{aligned}$$

□

Remark 18. For all function f, for all sets X_1 and X_2, $f|_{X_1}$ and $f|_{X_2}$ are always compatible.

Theorem 8. *If the union of the sets X_i is a super-set of the domain of a function f, then the union of the function f restricted by each set X_i is equal to the function f itself:* $(\mathrm{Dom}(f) \subseteq \bigcup_{i=1}^{n} X_i) \Rightarrow (\bigcup_{i=1}^{n} f|_{X_i} = f)$.

Proof.

$$\begin{aligned}
\bigcup_{i=1}^{n} f|_{X_i} &= \bigcup_{i=1}^{n} \big\{\big(x, f(x)\big)|x \in X_i \cap \mathrm{Dom}(f)\big\} \\
&= \left\{\big(x, f(x)\big)\middle| x \in \bigcup_{i=1}^{n} \big(X_i \cap \mathrm{Dom}(f)\big)\right\} \\
&= \left\{\big(x, f(x)\big)\middle| x \in \left(\bigcup_{i=1}^{n} X_i\right) \cap \mathrm{Dom}(f)\right\} \\
&= \big\{\big(x, f(x)\big)|x \in \mathrm{Dom}(f)\big\} = f
\end{aligned}$$

□

A.2. Redefinition of the projection

Remark 19. $src_{\pi_m(pg)}$, $dest_{\pi_m(pg)}$ and $properties_{\pi_m(pg)}$ can be redefined by using the restriction:

- $src_{\pi_m(pg)} = src_{pg}|_{\{m\}}$
- $dest_{\pi_m(pg)} = dest_{pg}|_{\{m\}}$
- $properties_{\pi_m(pg)} = properties_{pg}|_{\{(m,str)|str \in Str\}}$

Proof. For nodes, $m \in N_{pg}$ cannot be in the domain of src_{pg}, as their domain is a subset of E_{pg}. Therefore, $src_{pg}|_{\{m\}} = \emptyset \to \emptyset = src_{\pi_m(pg)}$.

For edges, $m \in E_{pg}$ is forced to be in the domain of src_{pg}, and its value is $src_{pg}(m)$. Therefore, $src_{pg}|_{\{m\}} = (m \mapsto src_{pg}(m)) = src_{\pi_m(pg)}$

The same reasoning applies for $dest_{pg}$.

The new definition of $properties_{\pi_m(pg)}$ that uses restrictions is immediate from the definition of the restriction. □

A.3. Proof of Theorem 5

Remark 20. The property graphs used in the following proof are described with formula. To help readability, for a given PG x, we allow ourselves to use the notation $N(x)$ instead of N_x. Similar notation will be used for $E(x)$, $src(x)$, $dest(x)$, $labels(x)$ and $properties(x)$. For example, $N_{\bigoplus_{m\in N_{pg}\cup E_{pg}}\pi_m(pg)}$ will instead of noted $N(\bigoplus_{m\in N_{pg}\cup E_{pg}}\pi_m(pg))$.

Proof. We first need to check if we can apply the $\oplus$ operator, i.e. if the three conditions of Definition 27 are met:

- When the π function is applied, nodes remain nodes and edges remain edges. The $\oplus$ operator also conserves this property. As $\forall m$, $N_{\pi_m(pg)} \subseteq N_{pg}$ and $E_{\pi_m(pg)} \subseteq E_{pg}$, the first condition is met.
- The definition of π (restriction of the original function), the definition of $\oplus$ (union of the functions) and the Lemma 6 (the union of two restriction is a restriction) imply that the *src*, *dest* and *properties* are compatible.

As $\oplus$ is commutative and associative, we can write the following decomposition: $\bigoplus_{m\in N_{pg}\cup E_{pg}}\pi_m(pg) = (\bigoplus_{m\in N_{pg}}\pi_m(pg)) \oplus (\bigoplus_{m\in E_{pg}}\pi_m(pg))$

To prove the theorem, we are going to check if it is true for all functions related to pg.

Edges (E_{pg}):

$$E\Big(\bigoplus_{m\in N_{pg}\cup E_{pg}}\pi_m(pg)\Big) = \bigcup_{m\in N_{pg}\cup E_{pg}} E\big(\pi_m(pg)\big) \qquad \text{[Definition of } \oplus \text{ on E]}$$

$$= \Big(\bigcup_{m\in N_{pg}} E\big(\pi_m(pg)\big)\Big) \cup \Big(\bigcup_{m\in E_{pg}} E\big(\pi_m(pg)\big)\Big)$$

$$= \Big(\bigcup_{m\in N_{pg}} \emptyset\Big) \cup \Big(\bigcup_{m\in E_{pg}} \{m\}\Big) = \bigcup_{m\in E_{pg}} \{m\} \qquad \text{[Definition of } \pi \text{ on E]}$$

$$= E_{pg}$$

Nodes (N_{pg}):

$$N\Big(\bigoplus_{m\in N_{pg}\cup E_{pg}}\pi_m(pg)\Big)$$

$$= \Big(\bigcup_{m\in N_{pg}} N\big(\pi_m(pg)\big)\Big) \cup \Big(\bigcup_{m\in E_{pg}} N\big(\pi_m(pg)\big)\Big)$$

$$= N_{pg} \cup \Big(\bigcup_{m\in E_{pg}} N\big(\pi_m(pg)\big)\Big)$$

To prove that the last expression above is equal to N_{pg}, we need to prove that $(\bigcup_{m\in E_{pg}} N(\pi_m(pg))) \subseteq N_{pg}$:

$$\forall m \in E_{pg}, \quad N\big(\pi_m(pg)\big) = \big\{src_{pg}(m), dest_{pg}(m)\big\} \subseteq N_{pg} \Rightarrow \bigcup_{m\in E_{pg}} N\big(\pi_m(pg)\big) \subseteq \bigcup_{m\in E_{pg}} N_{pg} = N_{pg}$$

Source of the edges (src_{pg}):

$$src\Big(\bigoplus_{m\in N_{pg}\cup E_{pg}}\pi_m(pg)\Big)$$

$$= \bigcup_{m \in N_{pg} \cup E_{pg}} src_{pg}|_{\{m\}}$$

$$= src_{pg} \qquad \left[\text{per Theorem 8, since } \bigcup_{m \in N_{pg} \cup E_{pg}} \{m\} \supseteq E_{pg} = \text{Dom}(src_{pg})\right]$$

Destination of the edges ($dest_{pg}$): The proof for $dest_{pg}$ follows the same steps as the proof for src_{pg}.

Properties ($properties_{pg}$) The proof is very similar to src_{pg}.

Noticing that:

- $\forall m \in N_{pg} \cup E_{pg}, properties(\pi_m(pg)) = properties_{pg}|_{\{(m,str)|str \in Str\}}$
- $\bigcup_{m \in N_{pg} \cup E_{pg}} \{(m, s)|s \in Str\} = \{(m, str)|m \in N_{pg} \cup E_{pg} \wedge str \in Str\} = (N_{pg} \cup E_{pg}) \times Str \supseteq \text{Dom}(properties_{pg})$

we can reapply the same reasoning as for src_{pg} to find

$$properties\left(\bigoplus_{m \in N_{pg} \cup E_{pg}} \pi_m(pg)\right) = properties_{pg}$$

Labels ($labels_{pg}$) The domain of definition of $labels(\bigoplus_{m \in N_{pg} \cup E_{pg}} \pi_m(pg))$ is:

$$N\left(\bigoplus_{m \in N_{pg} \cup E_{pg}} \pi_m(pg)\right) \cup E\left(\bigoplus_{m \in N_{pg} \cup E_{pg}} \pi_m(pg)\right) = N_{pg} \cup E_{pg}$$

The value of this function is $\forall x \in N_{pg} \cup E_{pg}$,

$$labels\left(\bigcup_{m \in N_{pg} \cup E_{pg}} \pi_m(pg)\right)(x) = \bigcup_{\substack{m \in N_{pg} \cup E_{pg} \\ \text{if } labels(\pi_m(pg))(x) \text{ is defined}}} labels\big(\pi_m(pg)\big)(x)$$

From the definition of π applied on *labels*, two outcomes are possible for $labels(\pi_m(pg))(x)$:

- For $m = x$, $labels(\pi_m(pg))(x) = labels_{pg}(x)$.
- For all other $m \neq x$, $labels(\pi_m(pg))(x)$ is either the empty set or undefined. In both cases, no extra value is contributed to $labels(\bigcup_{m \in N_{pg} \cup E_{pg}} \pi_m(pg))(x)$.

It can be concluded that $labels(\bigcup_{m \in N_{pg} \cup E_{pg}} \pi_m(pg))(x) = labels_{pg}(x)$, so $labels(\bigoplus_{m \in N_{pg} \cup E_{pg}} \pi_m(pg)) = labels_{pg}$.

Conclusion We have demonstrated that $\forall pg \in PGs$, $\forall f \in \{N_{pg}, E_{pg}, src_{pg}, dest_{pg}, labels_{pg}, properties_{pg}\}$, $f(pg) = f(\bigoplus_{m \in N_{pg} \cup E_{pg}} \pi_m(pg))$ therefore $\forall pg \in PGs, pg = \bigoplus_{m \in N_{pg} \cup E_{pg}} \pi_m(pg)$ □

References

[1] G. Abuoda, D. Dell'Aglio, A. Keen and K. Hose, Transforming RDF-star to Property Graphs: A preliminary analysis of transformation approaches, in: *Proceedings of the QuWeDa 2022: 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs Co-Located with 21st International Semantic Web Conference (ISWC 2022)*, Hangzhou, China, 23–27 October 2022, M. Saleem and A.N. Ngomo, eds, CEUR Workshop Proceedings, Vol. 3279, CEUR-WS.org, 2022, pp. 17–32, https://ceur-ws.org/Vol-3279/paper2.pdf.

[2] R. Angles, The property graph database model, in: *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management*, Cali, Colombia, May 21–25, 2018, D. Olteanu and B. Poblete, eds, CEUR Workshop Proceedings, Vol. 2100, CEUR-WS.org, 2018, https://ceur-ws.org/Vol-2100/paper26.pdf.

[3] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, A. Green, J. Hidders, B. Li, L. Libkin, V. Marsault, W. Martens, F. Murlak, S. Plantikow, O. Savkovic, M. Schmidt, J. Sequeda, S. Staworko, D. Tomaszuk, H. Voigt, D. Vrgoc, M. Wu and D. Zivkovic, PG-schema: Schemas for Property Graphs, *Proc. ACM Manag. Data* **1**(2) (2023), 198:1–198:25. doi:10.1145/3589778.

[4] R. Angles, A. Hogan, O. Lassila, C. Rojas, D. Schwabe, P. Szekely and D. Vrgoč, Multilayer graphs: A unified data model for graph databases, in: *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), GRADES-NDA '22*, Association for Computing Machinery, New York, NY, USA, 2022. ISBN 9781450393843. doi:10.1145/3534540.3534696.

[5] R. Angles, H. Thakkar and D. Tomaszuk, Mapping RDF databases to property graph databases, *IEEE Access* **8** (2020), 86091–86110. doi:10.1109/ACCESS.2020.2993117.

[6] G.A. Atemezing and A. Huynh, Knowledge Graph publication and browsing using Neo4J, in: *1st Workshop on Squaring the Circles on Graphs, SEMANTiCS*, Amsterdam, Netherlands, 2021.

[7] N. Beeren, Designing a Visual Tool for Property Graph Schema Extraction and Refinement: An Expert Study, CoRR, 2022, arXiv:2201.03643.

[8] A. Bonifati, S. Dumbrava, E. Martinez, F. Ghasemi, M. Jaffré, P. Luton and T. Pickles, DiscoPG: Property Graph schema discovery and exploration, *Proc. VLDB Endow.* **15**(12) (2022), 3654–3657, https://www.vldb.org/pvldb/vol15/p3654-bonifati.pdf. doi:10.14778/3554821.3554867.

[9] D. Brickley and R. Guha, RDF Schema 1.1, W3C Recommendation, W3C, 2014, https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[10] J. Bruyat, P.-A. Champin, L. Médini and F. Laforest, PREC: Semantic translation of property graphs, in: *1st Workshop on Squaring the Circles on Graphs, SEMANTiCS*, Amsterdam, Netherlands, 2021, https://hal.archives-ouvertes.fr/hal-03407785.

[11] H. Chiba, R. Yamanaka and S. Matsumoto, G2GML: Graph to graph mapping language for bridging RDF and Property Graphs, in: *Proceedings of the ISWC 2020 Demos and Industry Tracks: From Novel Ideas to Industrial Practice Co-Located with 19th International Semantic Web Conference (ISWC 2020), Globally Online*, November 1–6, 2020 (UTC), K.L. Taylor, R.S. Gonçalves, F. Lécué and J. Yan, eds, CEUR Workshop Proceedings, Vol. 2721, CEUR-WS.org, 2020, pp. 363–368, https://ceur-ws.org/Vol-2721/paper591.pdf.

[12] S. Das, J. Srinivasan, M. Perry, E.I. Chong and J. Banerjee, A tale of two graphs: Property Graphs as RDF in Oracle, in: *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014*, Athens, Greece, March 24–28, 2014, S. Amer-Yahia, V. Christophides, A. Kementsietsidis, M.N. Garofalakis, S. Idreos and V. Leroy, eds, OpenProceedings.org, 2014, pp. 762–773. doi:10.5441/002/EDBT.2014.82.

[13] T. Delva, J. Arenas-Guerrero, A. Iglesias-Molina, Ó. Corcho, D. Chaves-Fraga and A. Dimou, RML-star: A declarative mapping language for RDF-star generation, in: *Proceedings of the ISWC 2021 Posters, Demos and Industry Tracks: From Novel Ideas to Industrial Practice Co-Located with 20th International Semantic Web Conference (ISWC 2021), Virtual Conference*, October 24–28, 2021, O. Seneviratne, C. Pesquita, J. Sequeda and L. Etcheverry, eds, CEUR Workshop Proceedings, Vol. 2980, CEUR-WS.org, 2021, https://ceur-ws.org/Vol-2980/paper374.pdf.

[14] A. Dimou, M.V. Sande, P. Colpaert, R. Verborgh, E. Mannens and R.V. de Walle, RML: A generic language for integrated RDF mappings of heterogeneous data, in: *Proceedings of the Workshop on Linked Data on the Web Co-Located with the 23rd International World Wide Web Conference (WWW 2014)*, Seoul, Korea, April 8, 2014, C. Bizer, T. Heath, S. Auer and T. Berners-Lee, eds, CEUR Workshop Proceedings, Vol. 1184, CEUR-WS.org, 2014, https://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.

[15] N. Fathy, W. Gad, N. Badr and M. Hashem, ProGOMap: Automatic generation of mappings from Property Graphs to ontologies, *IEEE Access* **9** (2021), 113100–113116. doi:10.1109/ACCESS.2021.3104293.

[16] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer and A. Taylor, Cypher: An evolving query language for Property Graphs, in: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18*, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1433–1445. ISBN 9781450347037. doi:10.1145/3183713.3190657.

[17] O. Hartig, Foundations of RDF⋆ and SPARQL⋆ (an alternative approach to statement-level metadata in RDF), in: *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web*, Montevideo, Uruguay, June 7–9, 2017, J.L. Reutter and D. Srivastava, eds, CEUR Workshop Proceedings, Vol. 1912, CEUR-WS.org, 2017, https://ceur-ws.org/Vol-1912/paper12.pdf.

[18] O. Hartig, Foundations to query labeled Property Graphs using SPARQL, in: *Joint Proceedings of the 1st International Workshop on Semantics for Transport and the 1st International Workshop on Approaches for Making Data Interoperable Co-Located with 15th Semantics Conference (SEMANTiCS 2019)*, Karlsruhe, Germany, September 9, 2019, L. Kaffee, K.M. Endris, M. Vidal, M. Comerio, M. Sadeghi, D. Chaves-Fraga and P. Colpaert, eds, CEUR Workshop Proceedings, Vol. 2447, CEUR-WS.org, 2019, https://ceur-ws.org/Vol-2447/paper3.pdf.

[19] O. Hartig, P.-A. Champin, G. Kellogg and A. Seaborne, RDF-star and SPARQL-star, W3C Community Group Report, 2021, https://www.w3.org/2021/12/rdf-star.html.

[20] O. Hartig and B. Thompson, Foundations of an alternative approach to reification in RDF, 2014, arXiv preprint arXiv:1406.3399.

[21] S. Khayatbashi, S. Ferrada and O. Hartig, Converting property graphs to RDF: A preliminary study of the practical impact of different mappings, in: *GRADES-NDA '22: Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, Philadelphia, Pennsylvania, USA, 12 June 2022, V. Kalavri and S. Salihoglu, eds, ACM, 2022, pp. 10:1–10:9. doi:10.1145/3534540.3534695.

[22] D. Kontokostas and H. Knublauch, Shapes Constraint Language (SHACL), W3C Recommendation, W3C, 2017, https://www.w3.org/TR/2017/REC-shacl-20170720/.

[23] O. Lassila, M. Schmidt, O. Hartig, B. Bebee, D. Bechberger, W. Broekema, A. Khandelwal, K. Lawrence, C. López-Enríquez, R. Sharda and B.B. Thompson, The OneGraph vision: Challenges of breaking the graph model lock-in, *Semantic Web* **14**(1) (2023), 125–134. doi:10.3233/SW-223273.

[24] D.L. McGuinness, F. Van Harmelen et al., OWL web ontology language overview, *W3C recommendation* **10**(10), 2004, 2004.

[25] V. Nguyen, H.Y. Yip, H. Thakkar, Q. Li, E. Bolton and O. Bodenreider, Singleton Property Graph: Adding a Semantic Web abstraction layer to graph databases, in: *Proceedings of the Blockchain Enabled Semantic Web Workshop (BlockSW) and Contextualized Knowledge Graphs (CKG) Workshop Co-Located with the 18th International Semantic Web Conference, BlockSW/CKG@ISWC 2019*, Auckland, New Zealand, October 27, 2019, R. Samavi, M.P. Consens, S. Khatchadourian, V. Nguyen, A.P. Sheth, J.M. Giménez-García and H. Thakkar, eds, CEUR Workshop Proceedings, Vol. 2599, CEUR-WS.org, 2019, https://ceur-ws.org/Vol-2599/CKG2019_paper_4.pdf.

[26] F. Orlandi, D. Graux and D. O'Sullivan, Benchmarking RDF metadata representations: Reification, singleton property and RDF, in: *15th IEEE International Conference on Semantic Computing, ICSC 2021*, Laguna Hills, CA, USA, January 27–29, 2021, IEEE, 2021, pp. 233–240. doi:10.1109/ICSC50631.2021.00049.

[27] E. Prud'hommeaux and G. Carothers, RDF 1.1 Turtle, W3C Recommendation, W3C, 2014, https://www.w3.org/TR/2014/REC-turtle-20140225/.

[28] M.A. Rodriguez, The Gremlin graph traversal machine and language (invited talk), in: *Proceedings of the 15th Symposium on Database Programming Languages, DBPL 2015*, Association for Computing Machinery, New York, NY, USA, 2015, pp. 1–10. ISBN 9781450339025. doi:10.1145/2815072.2815073.

[29] M. Sporny, G. Kellogg and M. Lanthaler, JSON-LD 1.0, W3C Recommendation, W3C, 2014, https://www.w3.org/TR/2014/REC-json-ld-20140116/.

[30] S. Sundara, S. Das and R. Cyganiak, R2RML: RDB to RDF Mapping Language, W3C Recommendation, W3C, 2012, https://www.w3.org/TR/2012/REC-r2rml-20120927/.

[31] H. Thakkar, D. Punjani, J. Lehmann and S. Auer, Two for one: Querying property graph databases using SPARQL via gremlinator, in: *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, Houston, TX, USA, June 10, 2018, A. Arora, A. Bhattacharya, G.H.L. Fletcher, J.L. Larriba-Pey, S. Roy and R. West, eds, ACM, 2018, pp. 12:1–12:5. doi:10.1145/3210259.3210271.

[32] D. Tomaszuk, R. Angles and H. Thakkar, PGO: Describing property graphs in RDF, *IEEE Access* **8** (2020), 118355–118369. doi:10.1109/ACCESS.2020.3002018.

[33] O. van Rest, S. Hong, J. Kim, X. Meng and H. Chafi, PGQL: A property graph query language, in: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, GRADES '16*, Association for Computing Machinery, New York, NY, USA, 2016. ISBN 9781450347808. doi:10.1145/2960414.2960421.

[34] D. Wood, M. Lanthaler and R. Cyganiak, RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation, W3C, 2014, https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[35] R. Zhang, P. Liu, X. Guo, S. Li and X. Wang, A unified relational storage scheme for RDF and Property Graphs, in: *Web Information Systems and Applications*, W. Ni, X. Wang, W. Song and Y. Li, eds, Springer International Publishing, Cham, 2019, pp. 418–429. ISBN 978-3-030-30952-7. doi:10.1007/978-3-030-30952-7_41.