Semantic Web 0 (0) 1
IOS Press

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] *Université de Lyon, INSA Lyon, UCBL, LIRIS CNRS UMR 5205, Lyon, 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*

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 database model. In this model, 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 [1] and Gremlin [2] 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 [3], 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 [4]. In this model, data are represented with triples that represent links between resources. The resources and the links between them are identified through Internationalized Resource Identifiers (IRI). This data model is a W3C standard, and has been

---

[*] 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

studied through a large quantity of works, like RDFS [5] and OWL [6] for inference, or SHACL [7] for data validation. This model has been extended by RDF-star [8] that helps writing properties on triple terms in a more concise manner, but does not provide exactly the same modeling capabilities as PGs [9].

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. In [10], 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. Generally speaking, this diversity of graph models, and more precisely the lack of interoperability, hinders graph database adoption.

The foreseen scenario is the conversion from PG to RDF without information loss, so that users can modify their data and convert them back to the original PG model. We herein propose the reversible conversion part of this scenario. In a previous paper [11], 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 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[3], 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. The PRSC engine is available under the MIT licence[4] and can connect both to a Cypher endpoint and Gremlin endpoint.

The rest of this paper is organized as follows. Section 2 gives an overview on 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.

2. PRSC in practice

The Property Graph exposed on Figure 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 an RDF-star triple.

Using the user-defined mapping, PRSC is able to convert the PG in Figure 1 into the RDF-star graph in Listing 1, and more generally any Property Graph with the same schema into the corresponding RDF graph. The mapping the user must provide to the PRSC engine in Turtle-star format [12], and is exposed on Listing 2. 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:

---

[3] PG to RDF: Schema-driven Converter, pronounced “presque”

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

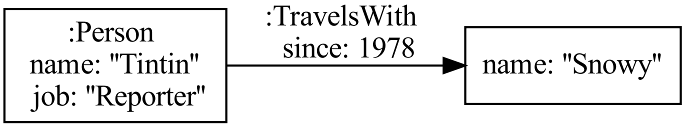

Fig. 1. A small PG about Tintin that serves as a running example in this paper

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

```
_:n1 rdf:type ex:Person .
_:n1 foaf:name "Tintin" .
_:n1 ex:profession "Reporter" .
_:n1 ex:isTeammateOf _:n2 .
<< _:n1 ex:isTeammateOf _:n2 >> ex:since 1978 .
_:n2 foaf:name "Snowy" .
```

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

```
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.
```

- 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.
    - ★ 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 in 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 merge.

Note that this mechanism of using quoted triples to describe templates in a pure Turtle-star file was already presented in our previous work [11]: compared to R2RML [13], it lets the user describe the triples to produce with a syntax closer to the triples that will actually be produced, but this process makes harder to express templated terms, for example a named node `<http://example.org/person/{name}>`, that users may want to use in subject position in which the `{name}` part is substituted with the value of the name property.

3. General definitions

This section introduce 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*” and “*Snowy*” are strings.

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

- $Dom(f) = \{x \mid \exists y \in A, \text{ such that } f(x) = y\}$

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

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

3.2. *Compatible functions*

For all functions f, we recall that they can be seen as sets: $f = \{(x, f(x)) \mid x \in Dom(f)\}$. For all sets S of 2-uples, 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 on Table 1.

Table 1

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

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

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:

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

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.* $\forall(x, y_f, y_g), (x, y_f) \in f \wedge (x, y_g) \in g \Rightarrow y_f = y_g$.

Remark 2. Two functions f and g are compatible iff $\forall x \in Dom(f) \cap Dom(g), f(x) = g(x)$.

3.3. *Property Graph*

Definition 3 (Property Graph)**.** Following the definition of Angles in [3], 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 list of nodes and the list of edges of the property graph PG.

- $src_{PG} : E_{PG} \rightarrow N_{PG}$ and $dest_{PG} : E_{PG} \rightarrow N_{PG}$ are two total functions. These two functions map each edge to its starting and destination nodes.
- $labels_{PG} : N_{PG} \cup E_{PG} \rightarrow 2^{Str}$ is a total function. This function maps the nodes and edges to their sets of labels.
- $properties_{PG} : (N_{PG} \cup E_{PG}) \times Str \rightarrow V$ is a partial function. This function describes the properties of the elements. V is the set of all possible property values.

The set of all property graphs is denoted PGs.

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

When notations are not ambiguous, we allow ourselves to omit the $_{PG}$ part. When the name of the graph is too long, the notation $N(PG)$ is used instead of N_{PG}.

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

- $N_{TT} = \{n_1, n_2\}\,; E_{TT} = \{e_1\}$
- $src_{TT} = \{e_1 \mapsto n_1\}\,; dest_{TT} = \{e_1 \mapsto n_2\}$
- $labels_{TT} = \{n_1 \mapsto \{\text{“}Person\text{”}\}\,; n_2 \mapsto \emptyset; e_1 \mapsto \{\text{“}TravelsWith\text{”}\}\}$
- $properties_{TT} = \left\{ \begin{matrix} (n_1, \text{“}name\text{”}) \mapsto \text{“}Tintin\text{”}; (n_1, \text{“}job\text{”}) \mapsto \text{“}Reporter\text{”} \\ (n_2, \text{“}name\text{”}) \mapsto \text{“}Snowy\text{”}; (e_1, \text{“}since\text{”}) \mapsto 1978 \end{matrix} \right\}$

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 $N_{PG_\emptyset} = E_{PG_\emptyset} = \emptyset$, $src_{PG_\emptyset} = dest_{PG_\emptyset} = labels_{PG_\emptyset} = \emptyset \rightarrow \emptyset$ and $properties_{PG_\emptyset} : \emptyset \times \emptyset \rightarrow \emptyset$.

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 is a bijective function $\phi : N_1 \cup E_1 \rightarrow N_2 \cup E_2$ where $\forall n \in N_1, \phi(n) \in N_2 \wedge \forall e \in E_1, \phi(e) \in E_2$.

Example 4. 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 G be a property graph and ϕ be a renaming function. The PG renaming function is defined as follows: $rename(\phi, G) = H$ with

- $N_H = \{\phi(n) \mid \exists x \in N_G\}$
- $E_H = \{\phi(e) \mid \exists e \in E_G\}$
- $src_H : e \in E_H \mapsto \phi(src_G(\phi^{-1}(e)))$
- $dest_H : e \in E_H \mapsto \phi(dest_G(\phi^{-1}(e)))$
- $labels_H : m \in N_H \cup E_H \mapsto labels_G(\phi^{-1}(m))$
- $properties_H : (m, prop) \in (N_H \cup E_H) \times Str \mapsto properties_G(\phi^{-1}(m), prop)$

Example 5. Let us consider TT, the PG about Tintin defined in example 3, and ϕ_{TT} the renaming function defined in the example 4.

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

- $N_{TT'} = \{a, b\}\,; E_{TT'} = \{c\}$
- $src_{TT'} = \{c \mapsto a\}\,; dest_{TT'} = \{c \mapsto b\}$
- $labels_{TT'} = \{a \mapsto \{\text{“}Person\text{”}\}\,; b \mapsto \emptyset; c \mapsto \{\text{“}TravelsWith\text{”}\}\}$
- $properties_{TT'} = \left\{ \begin{matrix} (a, \text{“}name\text{”}) \mapsto \text{“}Tintin\text{”}; (a, \text{“}job\text{”}) \mapsto \text{“}Reporter\text{”} \\ (b, \text{“}name\text{”}) \mapsto \text{“}Snowy\text{”}; (c, \text{“}since\text{”}) \mapsto 1978 \end{matrix} \right\}$

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#`

Definition 7 (Isomorphic property graph)**.** $\forall (G, H) \in PGs^2$, G and H are isomorphic iff $\exists \phi, rename(\phi, G) = H$

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

Existing works [1, 14] on query languages for PGs 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 on Table 2.

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

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[5] 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 $RdfTriples$.

A subset of $RdfTriples$ is an RDF graph.

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*.

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 on Listing 1 is composed of 5 triples written in Turtle format. In our formalism, the second triple, `_:tintin foaf:name "Tintin"`, is $(\_{:}tintin, foaf{:}name, \text{"}Tintin\text{"}^{xsd:string})$.

---

[5] For the sake of readability, although RDF-star is not yet part of the official RDF recommendation [4], 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.

3. $(ex{:}tintin, ex{:}travelsWith, ex{:}snowy)$ is an element of $RdfTriples$.
   $((ex{:}tintin, ex{:}travelsWith, ex{:}snowy), ex{:}since, \text{“1978”}^{xsd:integer})$ is an element of $RdfTriples$ that has a quoted triple in subject position.

Definition 10 (Term ownership)**.** 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 ownership examples)**.**

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

Definition 11 (List of blank nodes used in a graph)**.** $\forall G \subseteq RdfTriples, B_G$ is the list of blank nodes in the RDF graph G *i.e.* $B_G = \{bn \in B \mid \exists t \in G, bn \in t\}$.

Example 8. Let GTT be the RDF graph exposed on Listing 1. $B_{GTT} = \{\_{:}tintin, \_{:}snowy\}$

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.

4.1. Property graphs with blank nodes

Let us 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 $E \subset B$.

Furthermore, for any PG G, we can build an isomorphic PG H such that $N_H \cup E_H \subset B$. As discussed in Section 3.3.1, being isomorphic to G, H is indistinguishable from G for any practical purpose, because the exact *identity* of nodes and edges is not important: only the structure and the values of the PG is. Therefore without any loss of generality, we can restrict our work to PGs whose nodes and edges are elements of B.

Definition 12 (Blank Node Property Graph)**.** Let $BPGs$ be the set of property graphs with blank nodes only (BPG), *i.e.* $BPGs = \{G \in PGs \mid (N_G \cup E_G) \subseteq B\}$.

The PG to RDF graph conversion function that will be defined later is only defined for $BPGs$. Building a BPG isomorphic to the one that we want to actually convert, *i.e.* assigning to each node and edge a blank node and sticking to this choice for the duration of the conversion process, can be seen as the first step of the conversion.

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 $BTT = rename(\phi_{bTT}, TT)$:

- $N_{BTT} = \{\_{:}n1, \_{:}n2\}\,; E_{BTT} = \{\_{:}e1\}$
- $src_{BTT} = \{\_{:}e1 \mapsto \_{:}n1\}\,; dest_{TT'} = \{\_{:}e1 \mapsto \_{:}n2\}$
- $labels_{BTT} = \{\_{:}n1 \mapsto \{\text{“}Person\text{”}\}\,; \_{:}n2 \mapsto \emptyset; \_{:}e1 \mapsto \{\text{“}TravelsWith\text{”}\}\}$
- $properties_{BTT} = \left\{ \begin{matrix} (\_{:}n1, \text{“}name\text{”}) \mapsto \text{“}Tintin\text{”}; (\_{:}n1, \text{“}job\text{”}) \mapsto \text{“}Reporter\text{”} \\ (\_{:}n2, \text{“}name\text{”}) \mapsto \text{“}Snowy\text{”}; (\_{:}e1, \text{“}since\text{”}) \mapsto 1978 \end{matrix} \right\}$

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

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

Definition 14 (Type of a PG element)**.** A type is a triple $\in Types = (\{\text{“}node\text{”}, \text{“}edge\text{”}\} \times 2^{Str} \times 2^{Str})$.

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

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

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

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

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

Table 3

The types of the elements in the PG BTT

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

Remark 3. If two PGs F and G are isomorphic, their elements share the same type.

Indeed, by definition, $\exists \phi$ a renaming function from the elements of F to the elements of G, and $\forall m \in N_F \cup E_F, typeof_F(m) = typeof_G(\phi(m))$.

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 $?destination$[6].

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 $P = \{(l, valueOf) \mid l \in Str\}$. Elements of P can be noted with the same syntax as literals, for example $\text{“}name\text{”}^{valueOf}$ is the pair $(\text{“}name\text{”}, valueOf)$. 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:

- $\forall subject \in I \cup pvars, \forall predicate \in I, \forall object \in I \cup pvars \cup L \cup B, (subject, predicate, object) \in Templates$.
- $\forall tsubject \in Templates, \forall tobject \in Templates$, 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 $RdfTriples$, 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.

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

Example 11. The triple $(ex{:}tintin, rdf{:}type, ex{:}Person)$ is both an element of $RdfTriples$ and an element of $Templates$.

The triples $(?self, rdf{:}type, ex{:}Person)$ and $(ex{:}tintin, ex{:}name, \text{“}name\text{”}^{precValueOf})$ are members of $Templates$ but not of $RdfTriples$ because the first one uses an element of $?self$ and the second uses an element of P.

Definition 17 (Placeholders and template triple ownership)**.** The term ownership from Definition 10 is 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. Hence, an algorithm can retrieve in the context the template graph associated to each type of a PG and replace the placeholders of this template graph with data extracted from the PG.

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.*
$\forall type \in Dom(ctx)$:

- $\forall (key, valueOf) \in P, (\exists t \in ctx(type) \mid (key, valueOf) \in t) \Rightarrow key \in keys(type)$.
- $(kind(type) = \text{“}node\text{”}) \Rightarrow [\nexists t \in ctx(type) \mid ?source \in t \vee ?destination \in t]$.

---

[6] 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.

Intuitively, we use the placeholder literals of type $valueOf$ as placeholders for property values, so the used property keys must be in the type. $?source$ and $?destination$ are used as placeholders for the source and the destination nodes of an edge, so a node template should not use these values.

The set of all ctx functions is noted Ctx.

Definition 19 (Complete PRSC contexts for a given PG)**.** A PRSC context is said complete for a property graph $G \in BPGs$ iff there is a template graph defined for each type used in G. The set of all complete contexts for a PG G is noted $Ctx_G = \{ctx \in Ctx \mid \forall m \in N_G \cup E_G, typeof_G(m) \in Dom(ctx)\}$.

Example 12. Table 4 exposes an example of a complete ctx function for our running example.

Table 4

An example of a complete context for the Tintin Property Graph.

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

Example 13. The example exposed in Table 5 is not complete for the PG BTT as its domain lacks the type of $\_{:}n2$ and the type of $\_{:}e1$.

Table 5

An incomplete context for the Tintin PG

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

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

Table 6

A function that is not a context

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

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 $toLiteral : V \rightarrow L$, 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.

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

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

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

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

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

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

Table 7

Application of a PRSC context on the running example

b	$typeof(b)$	$ctx(typeof(b))$	$build(ctx(typeof(b)), PG, b)$
$\_:n1$	$(\text{“}node\text{”}, \{\text{“}Person\text{”}\},$ $\{\text{“}name\text{”}, \text{“}job\text{”}\})$	$(?self, rdf{:}type, ex{:}Person)$ $(?self, foaf{:}name, \text{“}name\text{”}^{valueOf})$ $(?self, ex{:}profession, \text{“}job\text{”}^{valueOf})$	$(\_:n1, rdf{:}type, ex{:}Person)$ $(\_:n1, foaf{:}name, \text{“}Tintin\text{”})$ $(\_:n1, ex{:}profession, \text{“}Reporter\text{”})$
$\_:n2$	$(\text{“}node\text{”}, \emptyset, \{\text{“}name\text{”}\})$	$(?self, foaf{:}name, \text{“}name\text{”}^{valueOf})$	$(\_:n2, foaf{:}name, \text{“}Snowy\text{”})$
$\_:e1$	$(\text{“}edge\text{”}, \{\text{“}TravelsWith\text{”}\},$ $\{\text{“}since\text{”}\})$	$(?source, ex{:}isTeammateOf, ?destination)$	$(\_:n1, ex{:}isTeammateOf, \_:n2)$

The resolution of n_2 is as follows:

$$\begin{aligned} & build(ctx(typeof(\_:n_2)), TT, \_:n_2) \\ = & build(ctx((\text{“}node\text{”}, \emptyset, \{\text{“}name\text{”}\})), TT, \_:n_2) \\ = & build(\{(pvar{:}self, foaf{:}name, \text{“}name\text{”}^{prec:valueOf})\}, TT, \_:n_2) \\ = & \{(\_:n_2, foaf{:}name, toLiteral(properties_{TT}(\_:n_2, \text{“}name\text{”})))\} \\ = & \{(\_:n2, foaf{:}name, toLiteral(\text{“}Snowy\text{”}))\} \end{aligned}$$

Algorithm 1: The *prsc* function

```
Input: $pg \in PG, ctx \in Ctx$
Output: An RDF graph
Main Function prsc($pg, ctx$):
    $rdf \leftarrow \{\}$
    forall element $m \in N_{pg} \cup E_{pg}$ do
        $tps \leftarrow ctx(typeof_{pg}(m))$
        /* build function */
        $built \leftarrow \{\}$
        forall $tp \in tps$ do
            $built \leftarrow built \cup \{\beta(tp, pg, m)\}$
        $rdf \leftarrow rdf \cup built$
    return $rdf$
/* In the formal definition, pg and m are implicitly passed to $\beta$ */
Function $\beta(t, pg, m)$:
    if $t \in Templates$ then
        $(s, p, o) \leftarrow t$
        return $(\beta(s, pg, m), \beta(p, pg, m), \beta(o, pg, m))$
    else if $t \in L$ then return $t$
    else if $t \in I$ then return $t$
    else if $t \in P$ then
        $(key, valueOf) \leftarrow t$
        return $properties_{pg}(m, key)$
    else
        $assert(t \in pvars)$
        switch $t$ do
            case $?self$ do return $m$
            case $?source$ do return $src_{pg}(m)$
            case $?destination$ do return $dest_{pg}(m)$
```

$$= \left\{ (\_{:}n2, foaf{:}name, ``Snowy"^{xsd:string}) \right\}$$

Algorithm 1 gives an algorithmic view of the *prsc* function.

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.

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 define the reversibility of a function f as the ability to find back x 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 in reasonable time. To illustrate this, 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 outputs 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 the function $prsc(\cdot, ctx) : pg \mapsto prsc(pg, ctx)$ is reversible.

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

Example 16 (A trivially non reversible context)**.** Consider $ctx_{\emptyset} \in Ctx$ such that $\forall type \in Types, ctx_{\emptyset}(type) = \emptyset$. $\forall G \in BPGs, prsc(G, ctx_{\emptyset}) = \emptyset$

As all PGs are mapped to the empty RDF graph, the use of the context $ctx_{\emptyset}$ makes the function $prsc$ not injective, and therefore not reversible.

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.

Definition 22 (Characterization function)**.**

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

$$\kappa(x) = \begin{cases} \bigcup_{triple \in x} \kappa(triple) & \text{if } x \subseteq RdfTriples \cup Templates \text{ (x is a graph or a template graph)} \\ \kappa(s) \times \kappa(p) \times \kappa(o) & \text{if } x = (s, p, o) \in RdfTriples \cup Templates \\ L & \text{if } x \in L \cup P \\ \{x\} & \text{if } x \in I \\ B & \text{if } x \in B \cup pvars \end{cases}$$

Remark 4 (κ on terms and triples is 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 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 PG $PG \in BPGs$ 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, $\forall t \in Templates,\ \kappa(\beta(t)) = \kappa(t)$

Example 17 (κ applied to the running example from Figure 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
  - $\kappa((\_{:}n1, foaf{:}name, \text{“}Tintin\text{”})) = \kappa((?self, foaf{:}name, \text{“}name\text{”}^{valueOf}))$
  - $(\_{:}n1, foaf{:}name, \text{“}Tintin\text{”}) \in \kappa((?self, foaf{:}name, \text{“}name\text{”}^{valueOf}))$
- $\kappa((?source, ex{:}isTeammateOf, ?destination)) = B \times \{ex{:}isTeammateOf\} \times B$

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

The idea behind the κ function is to map all wildcards to a common value to be able to check whenever we are able to distinguish the triples produced by different template triples with placeholders.

The κ function maps all templates to a super-set [7] 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 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.

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

Proof. (Sketch) This is a consequence of Remark 6 and the definition of *build*. □

Definition 23 (*unique* template triple)**.** The *unique* predicate determines if inside a set of template triples, a given template triple is the only one that can produce the triples it produces through *build*.

It is defined as follows with $t \in ts \subset Templates$:

$$unique(t, ts) = (\forall t' \in ts, \kappa(t) = \kappa(t') \Leftrightarrow t = t')$$

---

[7] 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.

Theorem 1 (Triples produced by a *unique* template triple)**.** 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 $tps = ctx(typeof_{pg}(m))$, $\forall td \in build(tps, pg, m), \forall tp \in tps$:

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

Proof. We prove the theorem by contradiction.

Let us suppose that:

- (A) $td \in build(tps, pg, b)$
- (B1) $unique(tp, tps) \triangleq (\forall t' \in tps, \kappa(tp) = \kappa(t') \Rightarrow tp = t')$
- (B2) $\kappa(td) = \kappa(tp)$
- (C) $td \notin build(\{tp\}, pg, b)$

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

As we reached a contradiction, it means that $td \in build(\{tp\}, pg, b)$.

□

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 conforms to those 3 criteria:

$\forall type \in Dom(ctx)$, let $tps = ctx(type)$

- *Element provenance*: all generated triples must contain the blank node that identifies the node or the edge it comes from.
    - $\forall t \in tps, ?self \in t$
- *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.
    - $\exists sign_{ctx}(type) \in tps, \forall x \in Dom(ctx), \kappa(sign_{ctx}(type)) \subseteq \kappa(ctx(x)) \Rightarrow x = type$
- *No value loss*: for all elements in the PG, we do not want to lose information stored in properties, and for edges, the source and destination node. Each of these pieces of information must be present in an unambiguously recognizable triple pattern.
    - $\forall key \in keys(type), \exists t \in tps \mid unique(t, tps) \wedge (key, valueOf) \in t$
    - $kind(type) = \text{“}edge\text{”} \Leftrightarrow \exists t \in tps \mid unique(t, tps) \wedge ?source \in t$
    - $kind(type) = \text{“}edge\text{”} \Leftrightarrow \exists t \in tps \mid unique(t, tps) \wedge ?destination \in t$

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

Remark 7 (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: $\forall tps \in Img(ctx), \exists t \in tps \Leftrightarrow \|tps\| \geqslant 1$.

Remark 8 (Inside a well-behaved context, each template graph is 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.* $\exists type1, type2$ 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 18. Table 8 studies the context used in our running example, exposed in Example 12.

Table 8

The running example context with the corresponding values through κ

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

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

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

Example 19 (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.

Table 9

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

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

This context is well-behaved:

- $?self$ appears in all triples,
- Template triples that are signature are marked with a ⋆. 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.

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

```
# From _:n1
_:n1 rdf:type ex:Person .
_:n1 foaf:name "Tintin" .
_:n1 ex:profession "Reporter" .
# From _:n2
_:n2 foaf:name "Snowy" .
_:n2 rdf:type ex:NamedEntity .
# From _:e1
_:e1 rdf:subject   _:n1 .
_:e1 rdf:object    _:n2 .
_:e1 rdf:predicate _:TravelsWith .
_:e1 ex:since      1978 .
```

Remark 9 (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:

$$\forall pg \in PGs, ctx \in Ctx^{+}_{pg}, \|prsc(pg, ctx)\| = 0 \Leftrightarrow pg = PG_{\emptyset}$$

Indeed, Remark 7 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.

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 4 steps algorithm: 1) it first assumes that the PG elements are all blank nodes in the RDF graph, 2) it gives a type to all elements with the $FindTypeOfElements$ function in Algorithm 3, 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.

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.

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: An element of $PGs$ or error
Main Function RDFToPG($rdf$, $ctx$):
    $Elements \leftarrow B_{rdf}$
    $typeof \leftarrow FindTypeOfElements(rdf, ctx, Elements)$
    $builtfrom \leftarrow AssociateTriplesWithElements(rdf, Elements, typeof)$
    return $buildpg(ctx, Elements, typeof, builtfrom)$
```

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

```
Input: $rdf \subset RDFTriples, ctx \in Ctx^+, Elements = B_{rdf}$
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 $t \in rdf \mid m \in t$ 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 $typeof$
```

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

```
Input: $rdf \subset RDFTriples, Elements = B_{rdf}, typeof : Elements \mapsto Type$
Output: A mapping $Elements \rightarrow 2^{RdfTriples}$ or error
Function $AssociateTriplesWithElements(rdf, Elements, typeof)$:
    $builtfrom \leftarrow \{\}$
    forall $b \in Elements$ do $builtfrom(b) \leftarrow \{\}$
    forall $td \in rdf$ do
        $bns \leftarrow \{term \in td \mid term \in B\}$
        if $(\exists! b \in bns)$ or $(\exists! b \in bns \mid kind(typeof(b)) =$ "edge"$)$ then
            $builtfrom(b) \leftarrow builtfrom(b) \cup \{td\}$
        else
            raise Error(No element provenance)
    return $builtfrom$
```

5.3.1. Finding the elements of the PG

The first step of the algorithm assumes 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)**.**

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

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 member of $PGs$ 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 $tp \in tps \mid unique(tp, tps) \land placeholder \in tp$ do
        $samekappa \leftarrow \{td \in tds \mid \kappa(td) = \kappa(tp)\}$
        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 $answer$
```

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 $B_{rdf} \subseteq N_h \cup E_h$.
- From Remark 7, we know that $ctx(typeof_{pg}(m))$ contains at least one triple pattern t. Combined with the *element provenance* criterion from Definition 24, we know that $?self \in t$. When *build* is applied to *tsign*, a triple that contains m is forced to appear, meaning that $N_{pg} \cup E_{pg} \subseteq B_{rdf}$.

□

Theorem 2 proves the correctness of the $Elements \leftarrow B_{rdf}$ 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 2. If a data triple shares the same value through κ as one of the signature triples of a type, then the element from which td was produced must be of this type:

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

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

$$\begin{aligned}
& \text{Assuming (A) } \kappa(td) = \kappa(sign_{ctx}(type)) \\
& td \in build(ctx(typeof(m)), pg, m) \\
\Rightarrow\ & \exists tp \in ctx(typeof(m)) \mid \kappa(td) = \kappa(tp) && \text{[Lemma 1]} \\
\Rightarrow\ & \exists tp \in ctx(typeof(m)) \mid \kappa(sign_{ctx}(type)) = \kappa(tp) && \text{[A]} \\
\Rightarrow\ & \exists tp \in ctx(typeof(m)) \mid \kappa(sign_{ctx}(type)) = \kappa(tp) \subseteq \kappa(ctx(typeof(m))) && \begin{bmatrix} tp \in ctx(typeof(m)) \\ \text{and by construction of } \kappa \end{bmatrix} \\
\Rightarrow\ & typeof(m) = type && \begin{bmatrix} \text{Signature template triple} \\ \text{in Definition 24} \end{bmatrix}
\end{aligned}$$

$\square$

Definition 25. The $candtypes_{nodes}$ and $candtypes_{edges}$, introduced in Algorithm 3, can be formally defined as:

$$\begin{aligned}
candtypes_{node}(b) = \{ & type \in Dom(ctx) \mid kind(type) = \text{“}node\text{”} \\
& \wedge \exists td \in rdf \mid b \in td \wedge \kappa(sign_{ctx}(type)) = \kappa(td)\} \\
candtypes_{edge}(b) = \{ & type \in Dom(ctx) \mid kind(type) = \text{“}edge\text{”} \\
& \wedge \exists td \in rdf \mid b \in td \wedge \kappa(sign_{ctx}(type)) = \kappa(td)\}
\end{aligned}$$

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)**.**

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

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

Table 10

A simple view of Theorem 3

	$\lVert candtypes_{node}(b) \rVert$	$\lVert candtypes_{edge}(b) \rVert$
$b \in N_{pg}$	1	any
$b \in E_{pg}$	0	1

Proof.

$$\forall b \in B_{rdf}, \forall type \in candtypes_{nodes}(b)$$

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

$$\begin{aligned}
& td \in rdf \\
\Leftrightarrow\ & td \in \bigcup_{m \in N_{pg} \cup E_{pg}} build(ctx(typeof(m)), pg, m) && [\text{Definition of } rdf \text{ / } prsc] \\
\Rightarrow\ & td \in \bigcup_{m \in N_{pg} \cup E_{pg} \mid typeof(m) = type} build(ctx(type), pg, m) && \begin{bmatrix} \kappa(td) = \kappa(sign_{ctx}(type)) \\ \text{and Lemma 2} \end{bmatrix} \\
\Rightarrow\ & td \in \bigcup_{m \in N_{pg} \mid typeof(m) = type} build(ctx(type), pg, m) && [kind(type) = \text{“}node\text{”}]
\end{aligned}$$

- We see that all triples td contributing to $candtype_{node}(b)$ 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 $ctx(typeof(b))$ must have generated a td containing b (since it must contain $?self$, according to Definition 24), so $typeof(b) \in candtype_{node}(b)$. Furthermore, no other node can produce a td containing b ($?self$ is the only blank node placeholder in node type templates), so $candtype_{node}(b)$ can not contain any other type. Therefore $candtype_{node}(b) = \{typeof(b)\}$.
- 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 as 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 $candtype_{edge} = \{typeof(b)\}$.

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 $candtype_{edge}(b)$ can contain an arbitrary number of types in that case. □

Remark 10. 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 cannot appear if the RDF graph was produced from a PG, making the $FindTypeOfElements$ function both sound and complete.

5.3.3. Finding the generated triples for each PG element

Theorem 4. In Algorithm 4, $\forall m \in N_{pg} \cup E_{pg}, build(ctx(typeof_{pg}(m), pg, m) = builtfrom[m]$.

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 the blank nodes in td as the potential elements m in the set bns is correct: the actual element m is in the set.

The Algorithm associates each triple td with a single $builtfrom[m]$:

- If there is only one distinct blank node m, it can only be produced by $build(ctx(typeof(m)), pg, m)$ so putting it in $builtfrom[m]$ is correct.
- If there are multiple distinct blank nodes:
  - $*$ Node template graphs only allow $?self$. No $m \in N_{pg}$ can produce td as it would require a template with at least two distinct members of $pvars$ value, which is impossible. So td cannot be generated from an m whose kind is “$node$”.

  - $*$ Edge template graphs allow all values of $pvars$. But only $?self$ can be mapped to edges, $?source$ and $?destination$ must be mapped to nodes. If there are multiple distinct blank nodes, at most one is an edge, the one mapped from $?self$ which is m, and the other ones can only be nodes. As the template graph is an edge template graph and m is the only edge in the triple, it is correct to put it in $builtfrom[m]$.
- If rdf was produced by $prsc$, it is impossible to reach the *Error(No element provenance)* case: at least one blank node is in the triple, 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

Cutting 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. To prove the correctness of the reversion algorithm, similar operators are needed for our formalization of PGs.

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

Let G be a PG.

Definition 26 (π projection of a Property Graph on an element)**.** $\forall m \in N_G \cup E_G$, $\pi_m(G)$ is a PG such as:

- If $m \in N_G$, $N_{\pi_m(G)} = \{m\}$, $E_{\pi_m(G)} = \emptyset$, $src_{\pi_m(G)} = dest_{\pi_m(G)} = \emptyset \to \emptyset$
- If $m \in E_G$, $N_{\pi_m(G)} = \{src_G(m), dest_G(m)\}$, $E_{\pi_m(G)} = \{m\}$, $src_{\pi_m(G)} = \{m \mapsto src_G(m)\}$, $dest_{\pi_m(G)} = \{m \mapsto dest_G(m)\}$
- $\forall x \in N_{\pi_m(G)} \cup E_{\pi_m(G)}, labels_{\pi_m(G)}(x) = \begin{cases} labels_G(x) & \text{if } x = m \\ \emptyset & \text{otherwise} \end{cases}$
- $\forall key \in keys(m), properties_{\pi_m(G)}(m, key) = properties_G(m, key)$, all other values are undefined.

Intuitively, 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.

Definition 27 (Property Graph merge operator $\oplus$)**.** We now define the $\oplus$ merge operator on property graphs. $\forall (A, B) \in PGs^2, \oplus(A, B)$ (or $A \oplus B$) is defined only if:

- $E_A \cap N_B = \emptyset \ \wedge \ N_A \cap E_B = \emptyset$
- src_A is compatible with src_B, $dest_A$ is compatible with $dest_B$ and $properties_A$ is compatible with $properties_B$. (see compatibility definition in Section 3.2).

Its value is $\oplus(A, B) = C$ with:

- $N_C = N_A \cup N_B$
- $E_C = E_A \cup E_B$
- $src_C : E_C \to N_C$, $src_C = src_A \cup src_B$.
- $dest_C : E_C \to N_C$, $dest_C = dest_A \cup dest_B$.
- $\forall m \in N_C \cup E_C, labels_C(m) = \begin{cases} labels_A(m) \cup labels_B(m) & \text{if both are defined} \\ labels_A(m) & \text{if only } labels_A(m) \text{ is defined} \\ labels_B(m) & \text{if only } labels_B(m) \text{ is defined} \end{cases}$
- $properties_C : (N_C \cup E_C) \times Str \to V$, $properties_C = properties_A \cup properties_B$.

Lemma 3. $\oplus$ 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 $\bigoplus$ merge of the π projection of a PG on all its elements is equal to the PG itself:

$$\forall G \in PGs, G = \bigoplus_{m \in N_G \cup E_G} \pi_m(G)$$

Proof. The proof is provided in Annex A. ☐

Use of cut The RDF graph built by $prsc$ from a PG pg is equal to $\bigcup_{m \in N_{pg} \cup E_{pg}} build(ctx(typeof_{pg}(m)), pg, m)$. 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.* $\forall k, properties_{pg}(m, k)$.
- If m is an edge, its source and destination nodes, *i.e.* $src_{pg}(m)$ and $dest_{pg}(m)$.

Therefore we can assert the following equality, $\forall pg \in PGs, \forall m \in N_{pg} \cup E_{pg}, \forall ctx \in Ctx_{pg}$:

$$\begin{aligned} & build(ctx(typeof_{pg}(m)), pg, m) \\ = \; & build(ctx(typeof_{pg}(m)), \pi_m(pg), m) \end{aligned}$$

$\pi_m(pg)$ can be considered as the minimal required Property Graph to produce the RDF triples related to the element m in the PG pg.

Proof of the algorithm Here, we prove the correctness of the $buildpg$ function in Algorithm 5.

Lemma 4. In Algorithm 5, at the end of an iteration of an element $b \in N_{pg} \cup E_{pg}$, the computed PG g_{after} is equal to $g_{before} \oplus \pi_b(pg)$, where g_{before} is the PG at the beginning of the iteration.

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.

Table 11

Description of the PG projection that is built in Algorithm 5

	$b \in N_{pg}$	$b \in E_{pg}$
$N(\pi_b(pg))$	$\{b\}$	$Img(src) \cup Img(dest)$
$E(\pi_b(pg))$	$\emptyset$	$\{b\}$
$src(\pi_b(pg))$	$\emptyset \to \emptyset$	$b \mapsto \boldsymbol{src_g[b]}$
$dest(\pi_b(pg))$	$\emptyset \to \emptyset$	$b \mapsto \boldsymbol{dest_g[b]}$
	$b \in N_{pg} \cup E_{pg}$	
$labels(\pi_b(pg))$	$\{b \mapsto labels(type)\}$	
$properties(\pi_b(pg))$	$\bigcup_{key \in keys(type)} \{(b, key) \mapsto \boldsymbol{properties_g[b, key]}\}$	

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 $key \in keys(typeof_{pg}(b))$ / $properties(b, key)$ are identical.

The $values$ set is filled by iterating on all tp such that $unique(tp, tps) \wedge ?source$. 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 property keys. 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 PG. 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 11 (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 4 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. The PG returned by Algorithm 5 is pg.

Proof. The PG g in the algorithm is initialized to $PG_\emptyset$. Lemma 4 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:

$$\begin{aligned}
g &= PG_\emptyset \oplus \bigoplus_{b \in N_{pg} \cup E_{pg}} \pi_m pg \\
&= \bigoplus_{b \in N_{pg} \cup E_{pg}} \pi_m pg && [PG_\emptyset \text{ is the neutral element of } \oplus] \\
&= pg && [\text{Theorem 5}]
\end{aligned}$$

□

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 $prsc^{-1}$ function for any well-behaved PRSC context *ctx*.

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 Figure 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 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:

Table 12

A context for the Tintin PG with the since property

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

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

```
% Tintin node
_:n1 rdf:type ex:Person .
_:n1 foaf:name "Tintin" .
_:n1 ex:profession "Reporter" .
% Snowy node
_:n2 foaf:name "Snowy" .
% TravelsWith edge
_:n1 ex:isTeammateOf _:n2 .
<< _:n1 ex:isTeammateOf _:n2 >> ex:since 1978 .
```

- $ctx(edgeunq)$ complies with the *no value loss* criterion and is not empty.
- For all triples $tp \in ctx(edgeunq)$:
  - $?source \in tp$ and $?destination \in tp$
  - tp is a *signature template triple*, *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:

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

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.

- $\forall pg1, pg2$ such that $prscEdgeUnique(pg1, ctx) = prscEdgeUnique(pg2, ctx)$, $pg1$ and $pg2$ are isomorphic.
- It is possible to define an algorithm such that $\forall pg$, from the RDF graph $prscEdgeUnique(pg)$, 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 of the value through κ of the triples of 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:

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

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

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

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.* $typeof(u) \in U$. 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 contains 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 [8, 15] propose to map each label to an RDF type or a literal used an 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 identifies the type by finding the unique signature template is not be 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 to represent 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 classic PG and RDF APIs. Angles et al. propose multilayered graphs [16], for which the OneGraph vision from Lasilla et al. [17] 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 [18], 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. [19] 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.[20].

From PGs to RDF In terms of PG to RDF conversion, the most impactful work is probably RDF-star [8, 9, 21, 22], 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 still merge 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. [23] study on a larger scale the different mappings described by Hartig and benchmark them, but they never consider using different modelings for different PG types during the same conversion. 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. To tackle the edge properties problem, Das et al. study how to use already existing reification techniques to represent properties [24]: 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) [15], 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[8] 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 [25] 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.

---

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

Instead of having a fixed mapping, our work on PREC [11] propose a mapping language named to drive the conversion from PG to RDF. Delva propose RML-star [26], an extension of RML [13] and R2RML [27] 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.

From RDF to PGs In Abuoda et al. [28] 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.

In [29], Angles et al. discuss different methods to transform an RDG graph into a PG. They propose different mappings, including an RDF-topology preserving one and a PG transformation. In [30], Atemezing and Hyunh 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 [31], 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 [32]. 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. [33] or Beereen [34]. 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.

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 [15]. 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 an adaptation of the reversion algorithm. It would need 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.* a reasoner, and then transform back the RDF graph into 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.

References

[1] 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*, 2018, pp. 1433–1445.

[2] M.A. Rodriguez, The gremlin graph traversal machine and language (invited talk), in: *Proceedings of the 15th Symposium on Database Programming Languages*, 2015, pp. 1–10.

[3] R. Angles, The Property Graph Database Model., in: *AMW*, 2018.

[4] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J.J. Carroll and B. McBride, RDF 1.1 concepts and abstract syntax, *W3C recommendation* **25**(02) (2014), 1–22.

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

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

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

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

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

[10] O. Lassila, M. Schmidt, B. Bebee, D. Bechberger, W. Broekema, A. Khandelwal, K. Lawrence, R. Sharda and B. Thompson, Graph? Yes! Which one? Help!, *arXiv preprint arXiv:2110.13348* (2021).

[11] 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.

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

[13] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens and R. Van de Walle, RML: a generic language for integrated RDF mappings of heterogeneous data, in: *Ldow*, 2014.

[14] 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*, 2016, pp. 1–6.

[15] D. Tomaszuk, R. Angles and H. Thakkar, Pgo: Describing property graphs in rdf, *IEEE Access* **8** (2020), 118355–118369.

[16] R. Angles, A. Hogan, O. Lassila, C. Rojas, D. Schwabe, P.A. Szekely and D. Vrgoc, Multilayer graphs: a unified data model for graph databases., in: *GRADES-NDA@ SIGMOD*, 2022, pp. 11–1.

[17] O. Lassila, M. Schmidt, O. Hartig, B. Bebee, D. Bechberger, W. Broekema, A. Khandelwal, K. Lawrence, C.M. Lopez Enriquez, R. Sharda et al., The OneGraph vision: Challenges of breaking the graph model lock-in 1, *Semantic Web* (2023), 1–10.

[18] R. Zhang, P. Liu, X. Guo, S. Li and X. Wang, A unified relational storage scheme for RDF and property graphs, in: *International Conference on Web Information Systems and Applications*, Springer, 2019, pp. 418–429.

[19] 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: *BlockSW/CKG@ ISWC*, 2019, pp. 1–13.

[20] F. Orlandi, D. Graux and D. O'Sullivan, Benchmarking RDF metadata representations: Reification, singleton property and RDF, in: *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*, IEEE, 2021, pp. 233–240.

[21] O. Hartig, Foundations of RDF* and SPARQL*:(An alternative approach to statement-level metadata in RDF), in: *AMW 2017 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017.*, Vol. 1912, Juan Reutter, Divesh Srivastava, 2017.

[22] O. Hartig, Foundations to Query Labeled Property Graphs using SPARQL., in: *SEM4TRA-AMAR@ SEMANTICS*, 2019.

[23] 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@ SIGMOD*, 2022, pp. 10–1.

[24] S. Das, J. Srinivasan, M. Perry, E.I. Chong and J. Banerjee, A Tale of Two Graphs: Property Graphs as RDF in Oracle., in: *EDBT*, 2014, pp. 762–773.

[25] 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)*, 2018, pp. 1–5.

[26] T. Delva, J. Arenas-Guerrero, A. Iglesias-Molina, O. Corcho, D. Chaves-Fraga and A. Dimou, RML-star: A declarative mapping language for RDF-star generation, in: *ISWC2021, the International Semantic Web Conference*, 2021, pp. 1–5.

[27] 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/.

[28] G. Abuoda, D. Dell'Aglio, A. Keen and K. Hose, Transforming RDF-star to Property Graphs: A Preliminary Analysis of Transformation Approaches–extended version, *arXiv preprint arXiv:2210.05781* (2022).

[29] R. Angles, H. Thakkar and D. Tomaszuk, Mapping RDF databases to property graph databases, *IEEE Access* **8** (2020), 86091–86110.

[30] G.A. Atemezing and A. Huynh, Knowledge Graph publication and browsing using Neo4J, in: *The 1st workshop on Squaring the circle on graphs*, 2021.

[31] H. Chiba, R. Yamanaka and S. Matsumoto, G2GML: Graph to graph mapping language for bridging RDF and property graphs, in: *International Semantic Web Conference*, Springer, 2020, pp. 160–175.

[32] A. Bonifati, S. Dumbrava, G. Fletcher, J. Hidders, B. Li, L. Libkin, W. Martens, F. Murlak, S. Plantikow, O. Savković et al., PG-Schema: Schemas for Property Graphs, *arXiv preprint arXiv:2211.10962* (2022).

[33] A. Bonifati, S. Dumbrava, E. Martinez, F. Ghasemi, M. Jaffré, P. Luton and T. Pickles, DiscoPG: property graph schema discovery and exploration, *Proceedings of the VLDB Endowment* **15**(12) (2022), 3654–3657.

[34] N. Beeren, Designing a Visual Tool for Property Graph Schema Extraction and Refinement: An Expert Study, *arXiv preprint arXiv:2201.03643* (2022).

Appendix A. Proof of properties on Property Graphs

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, $f|_X = \{(x, f(x)) \mid x \in X \cap Dom(f)\}$. $f|_X$ is called the restriction of the function f to the set X.

Remark 12. The restriction of a function to its domain is equal to the function itself:
$f|_{Dom(f)} = \{(x, f(x)) \mid x \in Dom(f)\} = f$.

Remark 13. A functional definition of Definition 29 would be, for all functions f, $f|_X : x \mapsto f(x)$ if $x \in X \cap Dom(f)$, undefined otherwise.

Lemma 5. $\forall$ functions f, sets X_1 and X_2, $f|_{X_1} \cup f|_{X_2} = f|_{X_1 \cup X_2}$.

Proof.

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

□

Remark 14. $\forall$ function f, sets X_1 and X_2, $f|_{X_1}$ and $f|_{X_2}$ are always compatible.

Theorem 8. $(Dom(f) \subseteq \bigcup_{i=1}^{n} X_i) \Rightarrow (\bigcup_{i=1}^{n} f|_{X_i} = f)$.

Proof.

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

□

A.2. Redefinition of the projection

Remark 15. $src_{\pi_m(G)}$, $dest_{\pi_m(G)}$ and $properties_{\pi_m(G)}$ can be redefined by using the restriction:

- $src_{\pi_m(G)} = src_G|_{\{m\}}$
- $dest_{\pi_m(G)} = dest_G|_{\{m\}}$
- $properties_{\pi_m(G)} = properties_G|_{\{(m,str) | str \in Str\}}$

Proof. For nodes, $m \in N_G$ cannot be in the domain of src_G, as their domain is a subset of E_G. Therefore, $src_G|_{\{m\}} = \emptyset \rightarrow \emptyset = src_{\pi_m(G)}$.

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

The same reasoning applies for $dest_G$.

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

A.3. Proof of Theorem 5

Proof. We first need to check if we can apply the $\oplus$ operator, i.e. if the two 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 $\forall m, N_{\pi_m(G)} \subseteq N_G$ and $E_{\pi_m(G)} \subseteq E_G$, 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 5 (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: $\bigoplus_{m \in N_G \cup E_G} \pi_m(G) = (\bigoplus_{m \in N_G} \pi_m(G)) \oplus (\bigoplus_{m \in E_G} \pi_m(G))$

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

Edges (E_G):

$$E(\bigoplus_{m \in N_G \cup E_G} \pi_m(G)) = \bigcup_{m \in N_G \cup E_G} E(\pi_m(G)) \qquad \text{[Definition of } \oplus \text{ on E]}$$

$$= (\bigcup_{m \in N_G} E(\pi_m(G))) \cup (\bigcup_{m \in E_G} E(\pi_m(G)))$$

$$= (\bigcup_{m \in N_G} \emptyset) \cup (\bigcup_{m \in E_G} \{m\}) = \bigcup_{m \in E_G} \{m\} \qquad \text{[Definition of } \pi \text{ on E]}$$

$$= E_G$$

Nodes (N_G):

$$N(\bigoplus_{m \in N_G \cup E_G} \pi_m(G))$$

$$= (\bigcup_{m \in N_G} N(\pi_m(G))) \cup (\bigcup_{m \in E_G} N(\pi_m(G)))$$

$$= N_G \cup (\bigcup_{m \in E_G} N(\pi_m(G)))$$

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

$$\forall m \in E_G, N(\pi_m(G)) = \{src_G(m), dest_G(m)\} \subseteq N_G$$
$$\Rightarrow \bigcup_{m\in E_G} N(\pi_m(G)) \subseteq \bigcup_{m\in E_G} N_G = N_G$$

Source of the edges (src_G):

$$\begin{aligned} & src(\bigoplus_{m\in N_G\cup E_G} \pi_m(G)) \\ = & \bigcup_{m\in N_G\cup E_G} src_G|_{\{m\}} \\ = & \; src_G \qquad \text{per Theorem 8, [since } \bigcup_{m\in N_G\cup E_G}\{m\} \supseteq E_G = Dom(src_G)] \end{aligned}$$

Destination of the edges ($dest_G$): The proof for $dest_G$ follows the same steps as the proof for src_G.

Properties ($properties_G$) The proof is very similar to src_G.
Noticing that:

- $\forall m \in N_G \cup E_G, properties(\pi_m(G)) = properties_G|_{\{(m,str) \mid str\in Str\}}$
- $\bigcup_{m\in N_G\cup E_G} \{(m, s) \mid s \in Str\} = \{(m, str) \mid m \in N_G \cup E_G \wedge str \in Str\} = (N_G \cup E_G) \times Str \supseteq Dom(properties_G)$

we can reapply the same reasoning as for src_G to find

$$properties(\bigoplus_{m\in N_G\cup E_G} \pi_m(G)) = properties_G$$

Labels ($labels_G$) The domain of definition of $labels(\bigoplus_{m\in N_G\cup E_G} \pi_m(G))$ is:

$$N(\bigoplus_{m\in N_G\cup E_G} \pi_m(G)) \cup E(\bigoplus_{m\in N_G\cup E_G} \pi_m(G)) = N_G \cup E_G$$

The value of this function is $\forall x \in N_G \cup E_G$,

$$labels(\bigcup_{m\in N_G\cup E_G} \pi_m(G))(x) = \bigcup_{\substack{m \in N_G \cup E_G \\ \text{if } labels(\pi_m(G))(x) \text{ is defined}}} labels(\pi_m(G))(x)$$

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

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

It can be concluded that $labels(\bigcup_{m\in N_G\cup E_G} \pi_m(G))(x) = labels_G(x)$, so $labels(\bigoplus_{m\in N_G\cup E_G} \pi_m(G)) = labels_G$.

Conclusion : We have demonstrated that
$\forall G \in PGs, \forall f \in \{N, E, src, dest, labels, properties\}, f(G) = f(\bigoplus_{m\in N_G\cup E_G} \pi_m(G))$ therefore $\forall G \in PGs, G = \bigoplus_{m\in N_G\cup E_G} \pi_m(G)$

□