N° d'ordre NNT : 2024*ISAL*0044

THESE de DOCTORAT DE L'INSA LYON, membre de l'Université de Lyon

Ecole Doctorale N° 512
Ecole Doctorale d'Informatique et de Mathématiques de Lyon

Discipline de doctorat : Informatique

Soutenue publiquement le 03/06/2024, par :

Julian Bruyat

Des graphes de propriétés aux graphes de connaissances

Devant le jury composé de :

LAMARRE Philippe	Professeur des Universités, INSA Lyon	Président
FARON Catherine	Professeure des Universités, Université Côte d'Azur	Rapporteure
HARTIG Olaf	Senior Associate Professor, Linköping University	Rapporteur
DIMOU Anastasia	Assistant Professor, Katholieke Universiteit Leuven	Examinatrice
LABRA GAYO Jose Emilio	Full Professor, University of Oviedo	Examinateur
LAFOREST Frédérique	Professeure des Universités, INSA Lyon	Directrice de thèse
CHAMPIN Pierre-Antoine	Maître de conférences - HDR, Université Lyon 1	Co-directeur de thèse
MÉDINI Lionel	Maître de conférences, Université Lyon 1	Co-directeur de thèse

Département FEDORA – INSA Lyon - Ecoles Doctorales

SIGLE	ECOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
ED 206 CHIMIE	**CHIMIE DE LYON** https://www.edchimie-lyon.fr Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage secretariat@edchimie-lyon.fr	**M. Stéphane DANIELE** C2P2-CPE LYON-UMR 5265 Bâtiment F308, BP 2077 43 Boulevard du 11 novembre 1918 69616 Villeurbanne directeur@edchimie-lyon.fr
ED 341 E2M2	**ÉVOLUTION, ÉCOSYSTÈME, MICROBIOLOGIE, MODÉLISATION** http://e2m2.universite-lyon.fr Sec. : Bénédicte LANZA Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 secretariat.e2m2@univ-lyon1.fr	**Mme Sandrine CHARLES** Université Claude Bernard Lyon 1 UFR Biosciences Bâtiment Mendel 43, boulevard du 11 Novembre 1918 69622 Villeurbanne CEDEX e2m2.codir@listes.univ-lyon1.fr
ED 205 EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTÉ** http://ediss.universite-lyon.fr Sec. : Bénédicte LANZA Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 secretariat.ediss@univ-lyon1.fr	**Mme Sylvie RICARD-BLUM** Laboratoire ICBMS - UMR 5246 CNRS - Université Lyon 1 Bâtiment Raulin - 2ème étage Nord 43 Boulevard du 11 novembre 1918 69622 Villeurbanne Cedex Tél : +33(0)4 72 44 82 32 sylvie.ricard-blum@univ-lyon1.fr
ED 34 EDML	**MATÉRIAUX DE LYON** http://ed34.universite-lyon.fr Sec. : Yann DE ORDENANA Tél : 04.72.18.62.44 yann.de-ordenana@ec-lyon.fr	**M. Stéphane BENAYOUN** Ecole Centrale de Lyon Laboratoire LTDS 36 avenue Guy de Collongue 69134 Ecully CEDEX Tél : 04.72.18.64.37 stephane.benayoun@ec-lyon.fr
ED 160 EEA	**ÉLECTRONIQUE, ÉLECTROTECHNIQUE, AUTOMATIQUE** https://edeea.universite-lyon.fr Sec. : Philomène TRECOURT Bâtiment Direction INSA Lyon Tél : 04.72.43.71.70 secretariat.edeea@insa-lyon.fr	**M. Philippe DELACHARTRE** INSA LYON Laboratoire CREATIS Bâtiment Blaise Pascal, 7 avenue Jean Capelle 69621 Villeurbanne CEDEX Tél : 04.72.43.88.63 philippe.delachartre@insa-lyon.fr
ED 512 INFOMATHS	**INFORMATIQUE ET MATHÉMATIQUES** http://edinfomaths.universite-lyon.fr Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage Tél : 04.72.43.80.46 infomaths@univ-lyon1.fr	**M. Hamamache KHEDDOUCI** Université Claude Bernard Lyon 1 Bât. Nautibus 43, Boulevard du 11 novembre 1918 69 622 Villeurbanne Cedex France Tél : 04.72.44.83.69 direction.infomaths@listes.univ-lyon1.fr
ED 162 MEGA	**MÉCANIQUE, ÉNERGÉTIQUE, GÉNIE CIVIL, ACOUSTIQUE** http://edmega.universite-lyon.fr Sec. : Philomène TRECOURT Tél : 04.72.43.71.70 Bâtiment Direction INSA Lyon mega@insa-lyon.fr	**M. Etienne PARIZET** INSA Lyon Laboratoire LVA Bâtiment St. Exupéry 25 bis av. Jean Capelle 69621 Villeurbanne CEDEX etienne.parizet@insa-lyon.fr
ED 483 ScSo	**ScSo**[1] https://edsciencessociales.universite-lyon.fr Sec. : Mélina FAVETON Tél : 04.78.69.77.79 melina.faveton@univ-lyon2.fr	**M. Bruno MILLY** (INSA : J.Y. TOUSSAINT) Univ. Lyon 2 Campus Berges du Rhône 18, quai Claude Bernard 69365 LYON CEDEX 07 Bureau BEL 319 bruno.milly@univ-lyon2.fr

[1] ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

Résumé

Les graphes de propriétés et les graphes RDF sont deux familles populaires de bases de données graphe. Bien qu'elles soient toutes les deux basées sur la notion de graphe, ces deux familles ne sont pas interopérables. Les graphes de propriétés constituent une famille d'implémentations de bases de données très flexible, où des propriétés peuvent être rattachées aux noeuds et aux arcs des graphes. La seconde est un modèle standardisé de description de connaissances, reposant sur des vocabulaires partagés entre tous les graphes RDF. Dans cette thèse, nous définissons des méthodes pour permettre une interopérabilité sémantique entre graphes de propriétés et graphes RDF, configurée à travers un "contexte" fourni par l'utilisateur. La première méthode est une méthode bas niveau, compatible avec n'importe quel graphe de propriétés. La seconde méthode est une méthode haut niveau, reposant sur la notion de schéma de graphe de propriétés, et pour laquelle la réversibilité de certains contextes est étudiée formellement. Enfin, pour faciliter l'écriture des "contextes" en RDF, et plus généralement de n'importe quel document RDF, nous proposons une méthode d'auto-complétion basée sur les vocabulaires de schémas RDF existants.

Abstract

Property graphs and RDF graphs are two popular categories of graph databases. However, despite the fact that they are both based on the notion of graphs, these two categories are not interoperable. Property graphs are a very flexible category of database implementations, where properties can be attached to the nodes and edges of the graph. The second is a standardized model for describing knowledge, based on vocabularies shared by all RDF graphs. In this thesis, we define methods to enable semantic interoperability between property graphs and RDF graphs, configured through a user-provided mapping named "context". The first method is a low-level method, compatible with any property graph. The second is a high-level method, based on the notion of property graph schema, and for which the reversibility of certain contexts is formally studied. Finally, to facilitate the writing of "contexts" in RDF, and more generally of any RDF document, we propose an auto-completion method based on existing RDF schema vocabularies.

Résumé substantiel

Introduction

Il existe de nombreux paradigmes de stockage de données. Avec le *big data*, des représentations NoSQL (Not Only SQL) se sont développées ; parmi elles les bases de données graphe. Dans une base de données graphe, les nœuds représentent les différentes entités, et les arcs les relations entre ces entités.

Au fil des années, deux manières de représenter des données sous forme de graphes se sont développées : les graphes de propriétés et les graphes RDF (Resource Description Framework).

Les graphes RDF sont un standard du W3C, publié en 1997 et depuis régulièrement mis à jour. Un graphe RDF est défini comme un ensemble de triplets RDF composé de trois termes : le sujet (le point de départ de l'arc), le prédicat (le type de relation décrit par l'arc) et l'objet (le point d'arrivée de l'arc). La majorité des termes dans un graphe RDF sont des URL, offrant à tous les graphes RDF une sémantique partagée car définie par le standard et le possesseur des URL utilisées. Les autres types de termes utilisables sont des nœuds blancs, des termes ayant une sémantique locale à un graphe RDF donné, et des littéraux, équivalent à des chaînes de caractères ne pouvant être utilisés qu'en position objet. Les graphes RDF étant définis formellement et ayant une sémantique partagée, de nombreux travaux se sont construits au-dessus de RDF, comme des ontologies ou des systèmes d'inférence.

Les graphes de propriétés sont une famille d'implémentations de graphes, dont une des plus connues est Neo4j, avec quelques différences mineures entre les différentes implémentations. On définit ici un graphe de propriétés comme étant un ensemble de nœuds et d'arc. Chaque nœud et chaque arc possède un ensemble d'étiquettes ainsi qu'un ensemble de propriétés, des paires clé-valeur.

Bien que le modèle des graphes RDF possède une sémantique bien définie, garantissant une compatibilité entre les différents graphes, les graphes de propriétés sont souvent considérés comme plus intuitifs, chaque nœud et arc pouvant stocker une multitude d'informations ; là où dans un graphe RDF, chaque information sera représentée sous la forme d'au moins un triplet / d'un arc. De plus, chaque moteur de graphe de propriétés possède souvent son écosystème complet, les rendant plus faciles à utiliser en apparence que l'écosystème RDF qui se compose d'une multitude d'outils dispersés. Néanmoins, choisir entre concevoir un graphe RDF ou un graphe de propriétés a des conséquences à long terme : choisir un modèle de graphe empêche l'utilisation des outils développés pour l'autre modèle.

Dans cette thèse, on souhaite proposer une interopérabilité sémantique entre les graphes de propriétés et les graphes RDF : l'ambition étant de proposer aux graphes de propriétés une manière d'obtenir une sémantique globale, les rendant compatibles avec les outils développés pour RDF. Cela passe non seulement par le fait de produire un graphe RDF à partir d'un graphe de propriétés, mais surtout de le rendre idiomatique : c'est-à-dire de produire un graphe RDF utilisant des constructions RDF usuelles et des ontologies déjà existantes qui peuvent être exploitées par des applications existantes.

Cette thèse présente deux contributions autour de l'interopérabilité entre graphes de propriétés et graphes RDF, articulées autour de convertisseurs qui sont pilotés par l'utilisateur via un graphe RDF nommé le contexte. Ce contexte décrit le *mapping* entre les termes du graphe de propriétés et la représentation RDF correspondante. Nous présentons également une troisième contribution avec une méthode d'auto-complétion de graphes RDF, visant à faciliter l'écriture des contextes.

État de l'art

Dans l'état de l'art, nous notons qu'il y a déjà des travaux de formalisation des deux modèles. En particulier, pour les graphes de propriétés, la définition formelle de Angles semble être celle qui fait aujourd'hui consensus. Des travaux existent déjà pour convertir d'autres formats vers RDF à travers une configuration fournie par l'utilisateur : R2RML pour les bases de données SQL, JSON-LD pour JSON ... C'est donc dans la continuité de ces travaux que nous proposerons une méthode similaire pour les graphes de propriétés. La particularité du problème de convertir des graphes de propriétés en graphe RDF est principalement dans la manière de représenter les propriétés des arcs des graphes de propriétés : en effet, dans un graphe RDF, un arc est défini comme un simple triplet, et ne peut donc pas contenir de propriétés. Certains auteurs étudient les différentes manières de représenter ces informations à travers différentes constructions comme la réification RDF classique ou les propriétés singletons comme Das et al. C'est dans ce sens que Hartig et Thompson ont développé RDF-star, une extension du modèle RDF permettant d'utiliser des triplets comme sujet ou objet d'autres triplets RDF. Cette extension, à la base développée pour l'interopérabilité entre les deux modèles de graphes, est jugée si intéressante qu'elle est actuellement en discussion pour être intégrée dans le standard RDF. D'autres auteurs proposent de convertir des graphes de propriétés en graphes RDF, soit en décrivant structurellement en RDF le graphe de propriétés comme le fait la *Property Graph Ontology*, soit en proposant une manière plus directe en forgeant de nouvelles URL pour les termes du graphe de propriétés comme le fait NeoSemantics, soit en demandant la correspondance entre les termes du graphe de propriétés et les URL comme le font Hartig et al. Mais si de nombreux travaux existent pour convertir des graphes de propriétés en graphe RDF, aucun ne tient réellement compte de la diversité des manières de représenter de l'information : dans un graphe RDF idiomatique, la manière de représenter le fait que deux personnes se connaissent n'est pas la même que celle pour représenter un contrat de travail par exemple.

Le framework PREC

La thèse commence par présenter les différents formalismes existants et à les discuter. Une formalisation alternative des graphes de propriétés est proposée afin de couvrir à la fois les graphes de propriétés répondant à la définition largement utilisée de Angles, et ceux supportés par Gremlin, une interface de requêtage populaire pour graphes de propriétés.

L'idée des graphes de propriétés à nœuds blancs est également introduite : lorsque l'on définit une fonction de conversion de graphes de propriétés vers graphes RDF, plutôt que de demander de fournir une fonction qui associe chaque élément du graphe de propriétés à un nœud blanc, la fonction de conversion attend un graphe de propriétés dont les éléments sont des nœuds blancs. Les éléments formels d'un graphe de propriétés ne portant aucune sémantique, deux graphes de propriétés isomorphes sont considérés égaux. Ainsi, le graphe de propriétés à nœuds blancs sera également une manière de prouver la réversibilité d'une fonction de conversion, en

montrant qu'il est possible de retrouver le même graphe de propriétés à nœuds blancs à partir d'un graphe RDF produit.

PREC-C : une conversion bas niveau

La première contribution est une conversion dite "bas niveau" des graphes de propriétés. L'utilisateur choisit une représentation par défaut des nœuds, arcs et propriétés, regroupés sous le terme de NEP (*Node, Edges and Properties*). Puis il peut spécialiser pour certains NEPs leur représentation en leur assignant une autre représentation.

Formellement, PREC-C est défini à travers une fonction qui prend en entrée un graphe de propriétés et un contexte PREC-C, et donne en sortie un graphe RDF. Un contexte PREC-C est défini comme une fonction totale associant à chaque sélecteur un graphe *template*. Un sélecteur permet de sélectionner les différents NEP selon leurs étiquettes (pour un nœud ou un arc) ou leur clé (pour une propriété), et le ou les NEPs auxquels ils sont rattachés (aucun pour les nœuds, les nœuds source et destination pour un arc, le porteur de la propriété pour une propriété). Un graphe *template* décrit la manière de représenter un NEP en RDF. Il contient des URL réservées faisant office de variables, qui seront remplacées par les valeurs propres à un NEP donné, comme un nœud blanc représentant le NEP, sa valeur pour une propriété, son nœud source ou son nœud destination pour un arc.

L'algorithme est présenté itérativement dans trois versions : la première n'utilise pas le contexte, produisant un graphe similaire à ce qui serait produit par la Property Graph Ontology. La seconde présente une implémentation naïve de l'application d'un contexte, en utilisant pour chaque NEP le graphe *template* correspondant à son sélecteur. La troisième version adapte la seconde version en tenant compte du fait qu'un graphe *template* peut modifier la manière dont un NEP portant une propriété est représenté : le plus souvent, soit par un nœud blanc, soit par un triplet qui sera à intégrer aux triplets de la propriété avec RDF-star.

Dans l'implémentation, les contextes sont décrits en RDF et l'utilisateur associe à des sous-ensembles de sélecteurs un template graphe. Ainsi, l'utilisateur fourni une représentation par défaut de chaque catégorie de NEP, puis spécialise la manière de représenter certains NEPs, par exemple comment représenter les arcs ayant une étiquette donnée.

Cette approche, si elle est capable d'émuler les méthodes de conversion existantes, souffre de deux problèmes majeurs. Ces problèmes viennent de la manière d'écrire des contextes PREC-C, en considérant les nœuds et les arcs indépendamment de leurs propriétés, le système refaisant ensuite le lien entre eux :

- les contextes sont souvent verbeux, en particulier lorsque l'on souhaite redéfinir toutes les propriétés ;
- il peut être difficile de prédire ce qu'il va produire, par exemple si pour un sélecteur donné, deux sous-ensembles de sélecteurs ont un graphe template redéfini mais qu'aucun n'est plus spécifique que l'autre ;
- il s'est avéré, dans des articles citant PREC-C, que l'approche était mal comprise, sans doute à cause de cette accumulation de différents concepts.

PRSC : une conversion basée sur les schémas

Dans cette approche, l'utilisateur fourni un contexte PRSC, qui fait la correspondance entre le schéma d'un graphe de propriétés, c.à.d. les différents types présents dans celui-ci, et les graphes *templates* décrivant comment les représenter en RDF.

Dans PRSC, un type de nœud ou d'arc ne concerne que ceux ayant strictement une liste d'étiquettes et de clés de propriétés données. Ainsi, dans PRSC, les propriétés sont contenues dans les types de nœuds et d'arc, et les graphes *templates* doivent décrire quel triplet produire pour chaque propriété du nœud ou de l'arc.

Lors de la conversion, pour chaque nœeud et arc, le convertisseur va calculer son type, chercher le graphe *template* correspondant dans le contexte, remplacer les espaces réservés en particulier pour les propriétés, ainsi que la source et la destination pour les arcs, et ainsi produire le graphe RDF correspondant.

La simplicité de cette nouvelle méthode de conversion nous permet alors d'en étudier les propriétés. En particulier, nous caractérisons une famille de contexte PRSC nommée les contextes PRSC bien élevés. Nous prouvons formellement que les conversions opérées à travers des contextes bien élevés sont réversibles : à partir du groupe RDF produit et du contexte PRSC bien élevé, il est possible de retrouver un graphe de propriétés isomorphe.

Un contexte PRSC bien élevé se repose sur trois critères :

- Chaque type est associé à une signature : un triplet *template* qu'il est le seul à produire.
- Le nœud blanc associé au nœud ou à l'arc est conservé dans chaque triplet.
- Chaque propriété, ainsi que pour les arcs, la source et la destination de l'arc, dispose dans le graphe *template* d'un triplet permettant de retrouver sa valeur de manière non ambiguë.

L'algorithme de reconstruction du graphe de propriétés se déroule ensuite en 4 étapes :

- Supposer que chaque nœud blanc du graphe RDF correspond à un nœud ou à un arc du graphe de propriétés.
- Pour chaque nœud ou arc, retrouver son type grâce aux triplets produits par les triplets signés.
- Isoler le sous-graphe RDF produit pour chaque nœud ou arc.
- Produire pour chaque sous-graphe RDF la partie correspondante du graphe de propriétés originel.

Les preuves formelles se reposent sur deux mécanismes majeurs :

- La définition d'une fonction de caractérisation κ qui associe chaque triplet *template* à un sur-ensemble des triplets qu'il peut produire.
- De nouveaux opérateurs formels sur les graphes de propriétés permettant de les décomposer et recomposer un graphe de propriétés selon ses éléments.

Des extensions sont proposées pour améliorer l'expressivité et la facilité d'usage de PRSC. En particulier, une extension est proposée pour gérer le cas où un type d'arc ne sera utilisé qu'une seule fois entre deux nœuds donnés : dans ce cas, la contrainte de représenter dans le graphe RDF le nœud blanc de l'arc est remplacée afin de faciliter la production de graphes RDF idiomatiques tout en conservant la preuve de la réversibilité de la conversion.

Shacled Turtle

Les contextes PREC-C et PRSC sont tous les deux décrits par l'utilisateur à travers un graphe RDF. Dans cette section, nous proposons une méthode pour tenter de faciliter non seulement l'écriture de contextes mais plus généralement l'écriture de graphes RDF qui sont conformes à une certaine ontologie.

L'intuition derrière Shacled Turtle est qu'une fois que l'on a donné le type d'une ressource RDF, lorsque l'on écrit de nouveaux triplets pour cette ressource, le moteur devrait suggérer les prédicats en lien avec les types de la ressource. Les ontologies comme RDFS et les schémas de validations comme SHACL sont des technologies qui décrivent précisément les liens entre les types et les prédicats qui leur sont liés ; et elles sont déjà couramment utilisées.

On propose donc une méthode pour convertir les différents triplets RDFS et SHACL en règles d'inférences et de suggestions et nous alimentons un moteur d'auto-complétion avec ces règles.

Néanmoins, lors d'une évaluation avec 30 utilisateurs, les résultats sont plutôt décevants : les utilisateurs ne prennent pas en compte le fait que le moteur leur suggère uniquement les prédicats qu'il a sélectionnés. À la place, ils se reposent principalement sur le fait de filtrer les termes en cherchant empiriquement à réduire la liste de termes proposés en tapant les premières lettres probables du terme qu'ils cherchent, et lorsqu'il ne reste que quelques termes, vérifient avec la description si le terme proposé correspond à celui qu'ils cherchent.

Des pistes d'améliorations de Shacled Turtle seraient donc de mettre en valeur les termes que le moteur considère pertinent, plutôt que d'enlever les autres afin de limiter la frustration des utilisateurs experts ; et d'utiliser l'analyse mise en place par le moteur pour compléter les descriptions des ontologies et expliquer pourquoi le moteur considère qu'un terme est plus pertinent qu'un autre.

Conclusion

Dans cette thèse, nous exposons deux manières de convertir des graphes de propriétés en graphes RDF à travers des deux types de contextes.

Les contextes PREC-C proposent une conversion très souple, permettant de représenter sous la forme de nœud blanc les propriétés, représenter les propriétés sur des propriétés, de convertir n'importe quel graphe de propriétés.

Les contextes PRSC se reposent sur un schéma, une liste de types, et ne peuvent convertir que les graphes de propriétés conformes à ce schéma. Cela permet de simplifier grandement l'algorithme de conversion, et nous permet d'étudier les propriétés des différentes catégories de contextes PRSC. Néanmoins, cette méthode perd la souplesse de PREC-C, en particulier en créant une profusion de types dans les cas où le type des différents nœuds et arcs sont très similaires à quelques étiquettes ou propriétés optionnelles prêt. À ce titre, il serait intéressant d'améliorer PRSC en lui apportant la puissance des propriétés PREC-C pour faciliter la gestion de propriétés optionnelles sans créer de nouveaux types dans le contexte.

De plus, si les propriétés formelles des contextes PRSC ont été étudiées, il serait intéressant de faire une étude avec des utilisateurs pour évaluer l'acceptabilité de l'approche, et comment l'améliorer au besoin.

Remerciements

Je vais commencer par remercier mes directeurs de thèse pour m'avoir fait confiance : Pierre-Antoine de m'avoir embarqué dans cette aventure, et Frédérique et Lionel pour m'avoir fait redescendre quand ça partait trop loin inutilement.

Merci également aux membres du jury d'avoir pris le temps de regarder mon travail : Philippe Lamarre qui a présidé le jury, Catherine Faron et Olaf Hartig qui ont rapporté et annoté mon travail, José Emilio Labra Gayo qui l'a reannoté, et Anastasia Dimou.

Merci également au LIRIS et à l'équipe TWEAK qui m'ont accueilli.

Merci à ma famille, sans qui je ne serais pas là : mes parents, ma soeur, mes cousins / cousines, oncles / tantes . . . , avec une pensée émue pour mon grand-père.

Merci à Jordan, Jonathan et Nuh de me supporter depuis tant d'années, surtout quand je pars dans des argumentations interminables sur des trucs sans intérêt. Merci également à Kenan qui m'a supporté mais dans un autre sens ; Alexandre, Eli, Théo, Gaétan, . . . ; Marc et Romain ; Stepehn, Gabriel, Walid, Julien, Jade, Amaury, Rémy ; Sylvain et Stéphane ; et ceux que j'ai oubliés.

Contents

Chapter 1

Introduction

Along the history of software engineering, multiple data modelling paradigms have been proposed. For a long time, tabular/relational databases, *i.e.* SQL (Structured Query Language) based databases, dominated the field. In this paradigm, multiple tables are defined: each table represents a category of information, for example a table of characters in comic books; each table has a list of columns, for example the name of the character, its gender, ... and each row is an entry, for example one line corresponds to the character Tintin, another to the character captain Haddock.

The rise of big data required the development of more flexible database types, following the NoSQL (Not Only SQL) paradigm. Multiple database systems have been developed, for example based on documents like MongoDB; key-value databases, like Redis; or graph databases.

Graph databases rely on the idea that data can be stored into a graph structure. Information is represented through nodes that represent objects and edges that represent the relationships between these objects.

Graph databases are also named knowledge graphs in the literature [1, 2]. Knowledge graph is a term created in the 1970s to denote a graph that is used to store knowledge about the world [3].

1.1 Two kinds of knowledge graphs

Two different types of graph databases have emerged: RDF (Resource Description Framework) graphs and Property Graphs.

1.1.1 RDF graphs

Listing 1.1: The RDF graph in Figure 1.1 in Turtle format

```
@prefix dbpedia:  <http://dbpedia.org/resource/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ex: <http://www.example.org/> .

# This first triple corresponds to the edge on the left
dbpedia:Titnin foaf:knows dbpedia:Captain_Haddock .

# This second triple corresponds to the edge on the right
dbpedia:Tintin foaf:name "Tintin" .

# Other triples
dbpedia:Tintin ex:owns "Snowy" .
dbpedia:Snowy foaf:name "Snowy" .
```

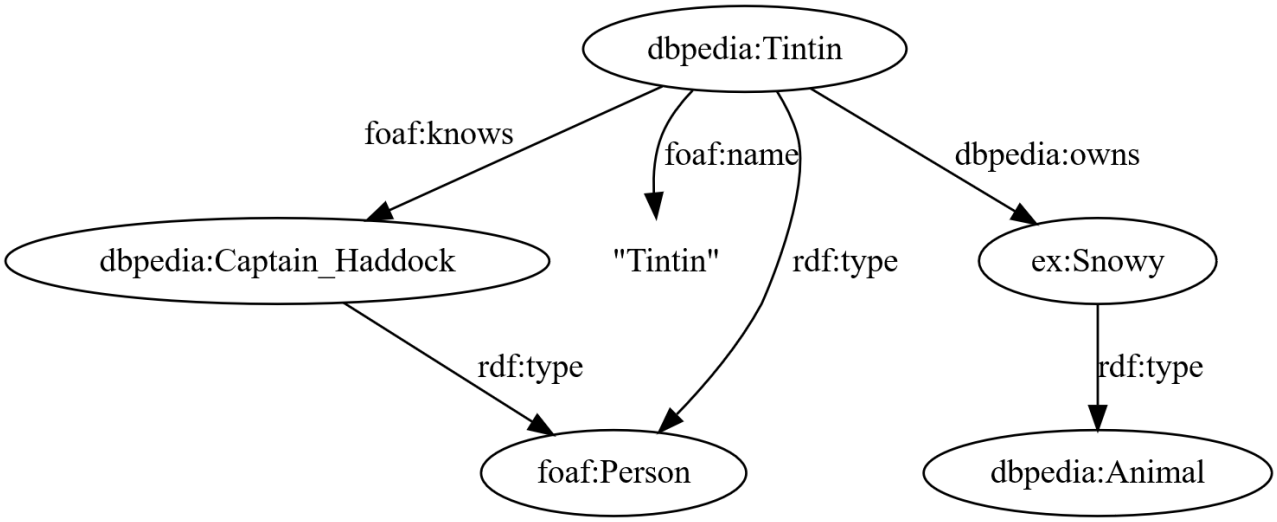


Figure 1.1: An example of an RDF graph about Tintin and the Captain Haddock

```
dbpedia:Tintin rdf:type foaf:Person .
dbpedia:Captain_Haddock rdf:type foaf:Person .
dbpedia:Snowy rdf:type dbpedia:Animal .
```

RDF graphs are specified in a W3C standard for which the initial draft has been published in 1997 by Lassila and Swick[1]. They are defined as a set of triples, describing each edge with the starting point of the edge named the *subject*, the ending point of the edge named the *object* and the type of relationship described by the edge named the *predicate*. For example, consider the RDF graph represented by a graphical representation in Figure 1.1 and a textual representation in Turtle [4] format in Listing 1.1. This RDF graph contains two triples. The first triple, on the left of the Figure, links a node named `dbpedia:Tintin` (the subject) to a node named `dbpedia:Captain_Haddock` (the object) and the edge is marked with the label `foaf:knows` (the predicate). This triple is usually described as the triple `dbpedia:Tintin foaf:knows dbpedia:Captain_Haddock`.

When the RDF model is used, resources are identified by using IRIs (Internationalized Resource Identifier, which can be seen as an extension of URLs). In particular, IRIs can be used in any position (subject, predicate or object). IRIs provide the RDF model with a semantic that is shared by all RDF graphs: the semantics of an IRI, *i.e.* what object it identifies, is determined by the owner of the IRI. In our example, the `dbpedia:` prefix is used as a shorthand for `http://dbpedia.org/resource/`. By consequence, the semantics of `dbpedia:Tintin` *i.e.* `http://dbpedia.org/resource/Tintin` is determined by the owner of the `http://dbpedia.org/` domain. Because IRI are heavily related to the web, and as RDF was designed for exchanging structured data on the web, the RDF community is often named the Semantic Web community: the purpose of RDF being to add semantics, *i.e.* machine-readable annotation, to a web that was at first developed for humans. In addition to IRIs, there are two other kinds of terms: blank nodes that can be considered as resources without an IRI, and literals which are a pair composed of a string and an IRI that explains how to interpret the string. In the RDF-star extension, there is a fourth kind of term that can be used in triples: the triplets themselves. Consider back the example of the RDF graph in Figure 1.1 and Listing 1.1. The subject of the second triple, on the right of the figure, is `dbpedia:Tintin`, its predicate is

[1] https://www.w3.org/TR/WD-rdf-syntax-971002/

`foaf:name` and the object is the *literal* "Tintin".

The RDF model also heavily relies on the concept of ontology, *i.e.* a common vocabulary that describes concepts, properties and relations in a certain domain. The terms of an RDF ontology typically share a common prefix, for example terms described by the foaf ontology share the `http://xmlns.com/foaf/0.1/` prefix, and users of the RDF model are heavily encouraged to use existing ontologies instead of creating new ones. Today, a wide diversity of ontologies exists, from very generic ontologies like the schema.org ontology that describes a wide variety of concepts, to more specialized ones like the foaf ontology that is focused on describing the relationships between people.

Besides being important for interoperability, ontology can bring inference capabilities to RDF graphs. Using (meta-)ontologies such as RDF Schema (RDFS) [5] and the Web Ontology Language (OWL) [6], one can formally describe the semantics of an ontology. Inference engines can then leverage these semantic descriptions to deduce additional triples from an RDF graph.

1.1.2 Property Graphs

Property Graphs (PGs) are a family of implementations of graph databases with no unique specification. It is usually accepted that modern Property Graphs were developed around 2010, with Neo4j[2] first release in 2010, and Apache TinkerPop[3] and the Gremlin traversal language[4] being first released in 2009.

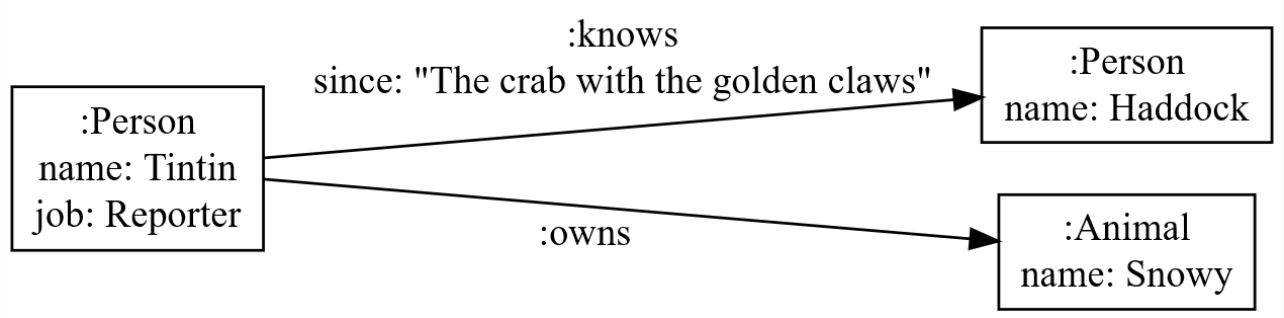


Figure 1.2: An example of a Property Graph about Tintin, Snowy and the Captain Haddock

PGs are usually described as a set of nodes and a set of edges that link these nodes. Labels are attached to nodes and edges to describe them. Some authors use the name "Labeled Property Graph" for PGs that allow the user to use labels [7, 8], which is the case of all modern PG implementations. For this reason, when we use the term PG, we actually mean Labeled Property Graph. Properties, a list of pairs with a property key and a property value, can also be added to the node and edges.

In Figure 1.2, we can see that there is a node with the `Person` label and two properties. The first property has the property key "name" and the property value "Tintin". The second property has the property key "job" and the property value "Reporter".

[2]https://neo4j.com/fr/
[3]https://tinkerpop.apache.org/
[4]https://tinkerpop.apache.org/gremlin.html

1.1.3 Choosing a knowledge graph model

RDF graphs are good candidates to be named knowledge graphs for several reasons: (1) They use a simple data model, consisting in of a set of subject-predicate-object triples, (2) they use IRIs for as-less-as possible ambiguous terms, (3) and they have a well-defined semantics, for example the fact that an RDF graph is monotonic (a triple can not contradict another triple), PGs could also be considered to be knowledge graphs as they also contain knowledge about a part of the world in the form of a graph.

When getting started in the project to build a graph database, developers have to choose between PGs or RDF graphs. Performance is not really a tiebreaker: some authors compare the performance of one system or another for querying data for their use cases [9, 10], without any clear winner for one system or the other. Usability is often more determinant. At first glance, the RDF ecosystem may look less practical: it is composed of a great variety of different standards, each one addressing a specific problem (defining the data model, specifying an inference system, querying data...). Moreover, the use of IRIs may not be intuitive at first, and the fact that the RDF model requires the use of an IRI for the different resources in the graph may look restrictive. On the other hand, getting started in building a PG is easier as the labels and properties of the nodes and edges of a PG are composed by raw strings and numbers, *i.e.* PGs have a local semantics. Moreover, PG engines generally come with intuitive tools, that encompass a lot of the features that a user expects (a querying language/API, a visualization, ...).

For example, when installing Neo4j, the database system comes with a Getting Started tutorial, the Neo4j application embeds a very intuitive graph visualization, Cypher, the querying language of Neo4j consists in "drawing the query in ASCII". On the other hand, SPARQL, the standard RDF query language uses pattern matching, which may seem less intuitive for

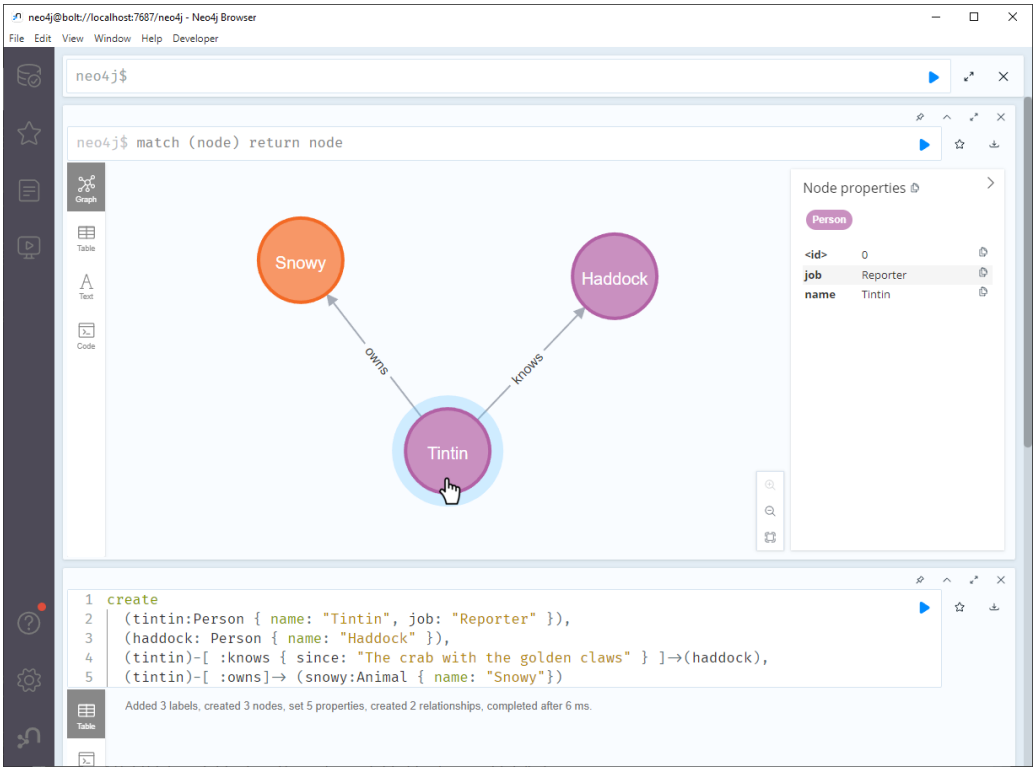


Figure 1.3: The Neo4j browser tool that comes with a Neo4j Desktop fresh install

some users. Figure 1.3 is a screenshot of the Neo4j Browser tool: 1) on the bottom, the running example Property Graph is created with a Cypher query, 2) on the top, by writing a query that queries all nodes, the whole PG is shown and can be interacted with. While some RDF database vendors also offer some kind of visualization, like GraphDB, overall, the RDF community recognizes that RDF may be harder to approach, as demonstrated by the existence of the EasierRDF initiative [5].

Despite the fact that RDF may seem to be a harder to use, its vast majority of tools and its well-defined semantics may sometimes miss to PG users. On the other hand, RDF users may sometimes want to use PG query languages and its tools to have a more compact visualization. Making a choice towards one system or the other has long term consequences. For example, Amazon Neptune is a database system that supports both systems, but forces the user to make a choice when creating their database between RDF or PG; users of Amazon Neptune often request the ability to change the database paradigm because they realize the other database system would actually better suit their use-cases [11].

1.2 Interoperability

Both RDF graphs and PGs have their advantages and disadvantages, as well as their advocates and detractors. Rather than being forced into choosing one or the other, all would benefit from increased interoperability between the two models.

Because both RDF graphs and PGs rely on a graph structure, one could think that converting a PG to an RDF graph and an RDF graph to a PG is a trivial task.

When studying the interoperability problem, we can distinguish two levels of interoperability:

- Syntactic interoperability is the ability of two systems to exchange data.
- Semantic interoperability is the ability of two systems to exchange the semantics of the data.

1.2.1 Syntactic interoperability

The key syntactic difference between PGs and RDF graphs lies in their structural representation. While a PG is actually defined as a graph, an RDF graph is not formalized as a graph but as a set of triples. Consequently, there are several structural differences:

- PGs allow adding properties to edges, while in RDF, there are no equivalents to adding properties to triples.
- Some PGs engine allow multiple edges with the same labels between the two same nodes, while in RDF, trying to produce an RDF graph with the same triple twice will collapse these triples into a single one (per the definition of a set of triples).

Syntactic interoperability has been addressed in multiple works, both from RDF graph to PGs [12][13] and from PGs to RDF graphs [14]. However, the solution proposed in these works produce graphs that heavily use the terminology of the other graph model. For example, from an RDF graph, these solutions produce a PG that uses the terms “IRI”, “literal” and “blank node” as labels of the produced PG, whereas labels are commonly expected to refer to concepts in the application domain.

[5] https://github.com/w3c/EasierRDF

1.2.2 Semantic interoperability

In addition to the syntactic differences between both models, there is also a semantic difference.

In a PG, the labels and property keys of nodes and edges are strings, and the values of the properties are strings, numbers, or array of these. The strings can use any arbitrary vocabulary, at the convenience of the designer of the PG. In other words, PGs use a *local semantics*: each PG has its own semantics and a term used in a PG can have another meaning in another PG. It may especially be the case for PGs in different languages: a "pain" label in an English PG likely refers to a medical condition, while in a French PG it likely refers to a bakery as "pain" means "bread". On the opposite, in the RDF model, all terms present in RDF graphs share the same semantics through the use of IRIs.

In addition to the semantics of the terms themselves, the RDF model also has its own semantics: for instance, the RDF model specifies that any RDF triple that is in an RDF graph is asserted, regardless of any other triple in the graph.

The main challenge to achieve semantic interoperability between the two models comes from this discrepancy between local and global semantics. In particular, converting a PG to an RDF graphs requires the local and largely implicit semantics of the PG to be elicited and aligned to RDF's global semantics. Existing tools, such as NeoSemantics, elude this issue by minting an ad-hoc ontology from the terms used in the PG. Others, such as PGO [14], merely encode the syntactic structure of the PG in RDF, leaving the domain-specific semantics implicit.

1.3 Motivation

The issue of PGs is not that they are not RDF graphs, but that the semantics stored in PGs is hidden in the lines of code of the applications that use them, and directly looking at the PG relies on the fact that the creator of the PG used a common language with the person that is currently looking at it. Moreover, as discussed earlier, because merging PGs is a tedious task, linking knowledge stored in multiple PGs is also a tedious task. In general, because the semantic used for PGs is a local semantics, a given PG is restricted to the applications that have been developed for it. On the opposite, consider a graph that uses terms that are shared with many other graphs, *i.e.* that use an explicit vocabulary/a widely shared vocabulary; such graph can be used in any application developed for these other graphs.

Hence, the goal of this thesis is to help users elicit the semantics of PGs. The method heavily relies on RDF as it is usually the accepted implementation of the knowledge graph idea: in practice, most knowledge graphs today are RDF graphs [1, 15, 16] like the WikiData knowledge graph [17]. By consequence, eliciting the semantics of a PG consists in defining how to convert it into RDF.

We therefore do not want to convert PGs to arbitrary RDF graphs containing the same information, but to actually produce an RDF graph that reuses existing terms and modelling patterns that are commonly admitted as RDF good practices. We name this kind of RDF graph that looks like other existing RDF graph *idiomatic RDF graphs*.

Listing 1.2: A non-idiomatic method to represent a "knows" relationship

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:e rdf:subject   _:alice .
_:e rdf:predicate foaf:knows .
_:e rdf:object    _:bob .
```

For example, consider the RDF graph in Listing 1.2. In this RDF graph, the fact that Alice knows Bob is represented using the RDF reification pattern, which is not the idiomatic method to assert that a person knows another one in RDF, especially using the `foaf:knows` predicate. On the opposite, an RDF graph that consists in a single triple, that tells that `_:alice foaf:knows _:bob` is an idiomatic RDF graph. This graph is idiomatic because the only namespace that is used is the `foaf` namespace, the "friend of a friend" being a very well known ontology, and it is used accordingly to how it is defined, *i.e.* the `foaf:knows` predicate is defined as a property in the ontology. Modelling a PG edge as an RDF triple is however not always the most idiomatic choice: an employment contract or any other temporally limited relationship should not be directly asserted in the graph, especially if the relationship is finished.

In addition to proposing methods to elicit the semantics through mapping languages, this thesis will also discuss the properties of the proposed mapping languages, both in terms of the capabilities of the mapping languages and desired properties of the mapping written by the user.

This process of defining mapping languages is very common to elicit data from other data models to RDF. The most known one is R2RML [18], a mapping language designed to produced RDF data from relational databases. Other mapping languages include RML [19], an extension of R2RML to any source and JSON-LD [20], a mapping language from the popular JSON format to RDF. All these tools make assumptions about the resources that are going to be represented: in R2RML, a row in a table corresponds to a resource; in RML, each object that comes from a logical source is a resource; and in JSON-LD the resources are expected to be JSON objects. In order to provide more expressivity to produce idiomatic RDF graphs, we propose mapping languages that are dedicated to PGs.

1.4 Structure of the thesis

The rest of the thesis is structured as follows: Chapter 2 gives an overview of the state of the art. The state of the art will be composed of two parts: one part focused on the PG to RDF graph conversion problem and one part with a higher focus on how RDF triples are produced. Chapter 3 gives an overview of the PREC framework under which the work of this thesis will be described, *i.e.* describe the formalism and the common parts of the PG to RDF methods described in the following chapters. In particular, the chapter introduces the concept of a context, an input provided by the user, in addition to the PG to convert, that will drive the conversion. Chapter 4 describes a first PG to RDF conversion method, named PREC-C. This conversion method is a highly customizable method, that is able to convert any supported PG from any PREC-C context. On the other hand, Chapter 5 describes a conversion method named PRSC in which contexts are based on the idea of a PG schema. In this chapter, we also exhibit some special properties of the PRSC schemas. Because both PREC-C and PRSC contexts are written in Turtle, in Chapter 6, we present Shacled Turtle, an auto-completion tool whose purpose is to help users writing RDF documents by using existing RDF ontologies and RDF schemas. Finally, in Chapter 7, we conclude with a discussion about the described converters, Shacled Turtle and potential future works.

Chapter 2

State of the art

In this chapter, different works related to PG and RDF graphs are discussed. In Section 2.1, PG and RDF graphs are presented under the scope of existing works. In Section 2.2, we discuss the existing works on RDF to PG conversion, and on PG to RDF conversion. This section serves as the base of the contributions of Chapters 3, 4 and 5 in which we propose our own solution to tackle the RDF to PG conversion problem. In Section 2.3, another field of the RDF domain is studied: the production of RDF triples and in particular tools that are built to help users writing them. The solution proposed for the PG to RDF conversion relies on a configuration file, written by the user in an RDF format. To make easier writing this configuration file, we propose an auto-completion engine that is presented in Chapter 6.

2.1 The two graph data models

2.1.1 Property Graphs

As mentioned in the introduction, PGs are not a unique model. Instead, they are a family of implementations. Some of the popular PG engines include Neo4j, Amazon Neptune, Azure Cosmos DB from Microsoft, or JanusGraph.

However, some kind of standardization work has been done, both at the theoretical level and at a practical level.

At the theoretical level, the consensual formal definition is the one provided by Angles [21] and which is used in multiple works on Property Graphs, in particular the one that focus interoperability between RDF and PGs. This definition will be provided later in the thesis in Definition 1 in Section 3.1 of Chapter 3. Today, all works are using Angles's definition or a definition that is very close. Intuitively, Angles's definition consists of defining a PG as a tuple composed of the set of nodes of the PG, the set of edges, one or two functions that maps the edges to its source and/or destination, one or two functions that maps the nodes and edges to its set of label and one or two partial functions that maps pairs composed of 1) a node or an edge and 2) a property key to the property value. Across existing works, the number and the name of these functions may differ slightly: for example, some works prefer to have a label function both for nodes and edges, while other works may use two different label functions.

At the practical level, Gremlin/Tinkerpop is a standard API for PG traversal. It has been developed so it can be used by many PG engines.

Some authors use a different method to formalize PGs. For example, Hölsch et al. [22] define the attributes (the equivalent of properties) as a set of functions. Each of these functions are defined for all nodes and/or edges and either return a value defined in a given set, or a special

value that means that this property does not exist for this node or edge. Bergami [23] proposes another definition, where each node and edge can be used as a function where the name of the attribute is passed as a parameter: for example, if a is a node and *name* is a property, $a(name)$ is the value of the "name" property. However, these formalisms are rare, and are either an answer for very specific needs, or are just older than Angles's definition.

GQL[1] is an ISO work whose purpose is to provide a SQL inspired query language, standard for all PG engines. The formal definition used in this work is the same as the one provided by Angles. SQL/PGQ is a subset of GQL designed for relational databases. It has been developed to be able to build views using data stored in a relational database and query them like a PG.

Junghanns et al. [24, 25] propose the Extended Property Graph Model (EPGM). In addition to a PG definition very close to Angles's one, the authors define a set of operators that can be used. However, the paper is mostly about a part of the implementation of Gradoop, a PG database system, so most operators are not formally defined.

Inspired by existing works on the classical join operator on SQL databases, Bergami et al. [26][23] propose a join operation defined on PGs. This work is mainly targeted to define how queries using the join operator can be defined. Hölsch et al. [22] propose different sets of operators to translate the different components of a Cypher query. This work is aimed at optimizing query plans by proposing a logic similar to the one developed to optimize relational queries. The common point of these works is that they are targeted at building queries. In this thesis, we will propose our own join operator, named the $\oplus$ merge operator. This operator will be designed to be able to work on PGs that may or may not share nodes and edges, including strange cases where an element is defined as a node in one graph and as an edge in the other. The particularity of this operator, and the π projection operator that will also be defined, is that they will be designed to be used in formal proofs.

Other authors translated the usual PG query languages to relational algebra. For example, Marton et al. formalized OpenCypher in [27], an open source version of the Cypher language developed for Neo4j. Thakkar et al. propose a formalization of the Gremlin API in [28].

2.1.2 RDF

RDF (Resource Description Framework) is a W3C standard designed for data exchange on the web. It has been designed to be decentralized and interoperable. To achieve this purpose, RDF has been designed to facilitate data integration by using global identifiers. The used global identifiers are IRIs, which have a shared semantics across all RDF graphs: the semantics of an IRI is specified by the owner of the IRI.

Formally, an RDF graph is defined as a set of triples of three elements. The first element is named the *subject*, the second the *predicate* and the third the *object*. As it is defined as a set, classic mathematical operators can be used; in particular the union $\cup$ operator can be used to merge two RDF graphs into a single one[2]. As any RDF graphs are supposed to be able to be merged, the RDF model is defined as monotonic model: adding or removing triples should not change the semantics of other triples.

RDF has been standardized in 1999 [29], and has been updated multiple times, in 2004 [30], in 2014 [31], and a new update is being currently discussed based on the work of the RDF-star W3C Community Group [32]. A formal definition of RDF will be provided in Chapter 3. This

[1] https://www.gqlstandards.org/ / https://www.iso.org/standard/76120.html

[2] Only in the case where the two RDF graphs do not share any blank node. Otherwise, the merge operation is specified by 1) first computing RDF graphs that are isomorphic to the two RDF graphs to merge and that share no blank nodes, 2) and then the $\cup$ operator can be applied.

formal definition will include quoted triples/RDF-star.

Note that, unlike a mathematical graph or PG, an RDF graph is not defined a set of nodes and a set of edges, but as a set of triples. This has several subtle consequences. For example, while in PGs, a node with no edges can exist, in an RDF graph, to exist, a node must either have an out-coming edge (being in the subject position of a triple) or have an incoming edge (being in the object position of a triple); the predicate of the triple serving as the label of the edge. Other differences include the fact that between two RDF nodes, multiple edges can not share the same IRI/label, while in PGs this feature is supported.

As the data model is very simple, patterns to represent composite data have been developed. A famous such pattern is the standard RDF reification, described in the RDF standard [33] in the "Reification" section as a method to add knowledge about RDF triples, even about the ones that are not in the RDF graph[3]. The RDF reification consists in assigning to an RDF resource the semantics of a triple, adding three triples to describe the subject, the predicate and the object of the triple assigned to the resource.

Listing 2.1 gives an example of the use of the RDF reification pattern to add the information about the fact that Tintin travels with the Captain Haddock since "The Crab with the Golden Claws". Triples are added to denote that the blank node $\_:a$ identifies the RDF triple "Tintin travels with Captain Haddock" and the temporal meta-data is added by adding a new triple with $\_:a$ in the subject position. Listing 2.2 lists the same information, but instead of using an RDF reification, thanks to RDF-star, the triple "Tintin travels with Captain Haddock" itself can be used in place of $\_:a$. Note that in both examples, the triple is also part of the graph because in this case, the annotated triple is also considered true.

Listing 2.1: An example of the RDF reification pattern in Turtle

```
# An RDF triple that tells that Tintin travels with the Captain Haddock
dbpedia:Tintin ex:travelsWith dbpedia:Captain_Haddock .

# The RDF reification is used to ad
## The blank node _:a identifies the RDF triple above
_:a rdf:type rdf:Statement .
_:a rdf:subject dbpedia:Tintin .
_:a rdf:predicate ex:travelsWith .
_:a rdf:object dbpedia:Captain_Haddock .

## Add a temporal information about the triple through the use of _:a
_:a ex:since ex:TheCrabWithTheGoldenClaws .
ex:TheCrabWithTheGoldenClaws schema:name "The Crab with the Golden Claws" .
```

Listing 2.2: An example of how RDF-star can be used in place of the RDF reification in Turtle-star

```
# An RDF triple that tells that Tintin travels with the Captain Haddock
dbpedia:Tintin ex:travelsWith dbpedia:Captain_Haddock .

# The metadata is added to the triple itself by using it as the subject
<< dbpedia:Tintin ex:travelsWith dbpedia:Captain_Haddock >>
  ex:since ex:TheCrabWithTheGoldenClaws .

ex:TheCrabWithTheGoldenClaws schema:name "The Crab with the Golden Claws" .
```

Multiple other constructs have been proposed to represent RDF data. These constructs are grouped under the term "ontology design patterns", are listed by some platforms[4], and are

[3]Because RDF graphs are monotonic, an RDF triple can not be in the RDF graph if the triple is not considered true. The RDF reification provides an option to comment that a given triple may only be true with a certain confidence percentage, or that it is false.

[4]For example http://ontologydesignpatterns.org

frequently the topic of workshops.

As RDF has been standardized for a long time, in addition to ontology design patterns, a lot of specifications have been developed around RDF: SPARQL [34] is a standard query language for querying RDF data, multiple specifications for reasoning have been developed like RDFS [5] and OWL [6], specifications for validating data like SHACL [35] and ShEx[5]... RDF also has a variety of serialization formats, among which RDF/XML [36], Turtle [4] and JSON-LD [20]. Among these tools, two kinds will be the starting point of the work in Chapter 6: reasoning systems and validation engines:

Ontologies and reasoning systems Gruber defines an ontology as a specification of a conceptualization of the concepts and relationships between these concepts in an area of interest [37].

RDFS [38] (RDF schema) is a popular and simple ontology language. It provides a way to define classes and properties in RDF, organizing them into specialization hierarchies/lattices, and relating properties to their domain and range.

Another popular ontology language is OWL [6] (Web Ontology Language). Compared to RDFS, it includes more properties to describe ontologies, and is much more expressive. However, as the full OWL semantics is undecidable, it has been specialized into several profiles, with different expressiveness and associated time complexity.

Validating schemas SHACL [35] (Shapes Constraint Language) is a W3C specification to check RDF graph validity against a set of constraints. A SHACL graph describes shapes and the set of constraints associated to these shapes. When a resource is targeted by a shape, the SHACL validator will check if none of the associated constraints are violated.

ShEx (Shape Expressions) is another validating schema language developed for RDF. Compared to SHACL, ShEx takes a different approach to validation, but discussing it is out of scope of this thesis. Furthermore, ShEx shapes are expressed in an ad-hoc compact grammar, while SHACL shapes are expressed in RDF.

Converting data to RDF To produce RDF data, mapping data from one format to RDF has been a widely studied problem, mostly from SQL and spreadsheets.

R2RML [18] maps relational databases and tabular data to RDF by using mappings provided by the user, RML [19] extends the latter to support other kinds of data sources, RDF123 [39] aims to produce RDF data by using spreadsheets as an abstraction, JSON-LD [20] transforms JSON documents to RDF and is the way recommended by Google to add metadata to a website in order to improve its SEO (Search Engine Optimization). The focus of the thesis is to convert PG data into RDF, and the following Section 2.2 will focus more towards PG and RDF graph interoperability.

When designing a mapping language, users should be able to describe the RDF triples that they want to model, multiple templating systems already exist:

- An R2RML/RML document is a set of triple maps. Each triple map has a logical table/logical source as the object of `rr:logicalTable`/`rml:logicalSource`, and the set of triples to produce as the value of `rr:subjectMap`, `rr:predicateObjectMap`, `rr:predicateMap` and `rr:objectMap`. The object of the four latter predicates can either be the term to use, or a term that describes how to construct the term to use from the input data. The system enables to use IRIs composed of multiple values from the logical source

[5] http://shex.io/shex-semantics/index.html

at any position. However, this system is very verbose, requiring to write at least four triples to produce one triple.

- OTTR [40] is a tool to produce RDF graphs and ontologies. It offers a templating system that requires the user to provide a list of parameters that describes the input of the OTTR template and a pattern that describes the triples to produce. The OTTR template can then be instantiated to produce the triples. OTTR templates and instantiations can be serialized in multiple formats, including a custom format and an RDF serialization. However, in its Turtle serialization, all templates, including the OTTR template to describe a single RDF triple, the `ottr:Triple` template, take a list of terms as a parameter which may be error-prone in the case of the `ottr:Triple` template. Quoted triples are also not supported by the system.
- SPARQL-Generate [41] is an extension of SPARQL to produce RDF data from heterogeneous sources. SPARQL-Generate offers powerful constructs to build terms from data, iterate on data and describe the triples to produce. However, being an extension of SPARQL, SPARQL-Generate introduces new keywords to learn for the user and support for the implementation, even considering that the authors describe the learning curve as very low for a user that already knows SPARQL. In this thesis, we wanted to evaluate the opportunity of using quoted triples from RDF-star as a simple solution to describe triples to produce in use-cases in which powerful constructs are not required.

2.2 Comparing RDF with PGs

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

Opposing one family of graphs to another In *Semantic property graph for scalable knowledge graph analytics*, S. Purohit et al. [42] propose to store RDF graphs into PGs which are named **Semantic Property Graphs**. In particular, they map RDF reified triples to PG edges. The authors argue that as the resulting Semantic Property Graph is smaller than the original RDF graph, in the sense that there are fewer nodes, data analytics is easier to perform. While this work propose a way to represent RDF data in PGs, the produced PGs are not intended to be merged with other existing PGs.

On the opposite, Alocci et al. [9] stored the structure of a particular molecule, the Glycan molecule that they were studying, in several RDF graph database engines and in Neo4j in order to compare their performances. All measured RDF engines have equivalent benchmarks expect Blazegraph which is way faster.

Warren and Mulholland [43] compared PG and RDF from a usability perspective. They note that there are very few work about it. They proposed different modelings both in RDF and Cypher to participants. Participants considered that both PGs and RDF graphs were as easy to use. The modelling decisions made by the participants both in PGs and RDF graph were similar, in particular in terms of creating a new node for some resources like cities instead of assigning them to properties. However, the researchers note that most participants were by default more familiar with RDF than PGs which may induce a bias in these modelling choices.

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 [44], of which the **OneGraph** vision from Lassila et al. [11] 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, Lassila et al. 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 [45], 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. [46] 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. [47].

From PGs to RDF In terms of PG to RDF conversion, the most impactful work is probably RDF-star [48, 49, 7, 32], 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. With RDF-star, it is possible to use RDF triples in the subject or object position of other RDF triples. However, most PG engines support multi-edges, *i.e.* two edges of the same type between the two same nodes. On the other hand, the naive approach consisting in using the source node, the type of the edge and the destination node as respectively the subject, the predicate and the object of an RDF triple would collapse the multi-edges. Converting each edge property to an RDF-star triple that uses the former triple as its subject would lead to the properties of each multi-edge to be merged. Even outside the context of converting PGs to RDF graph, the RDF-star proposal is so impactful that it is the new key feature that will be fully integrated in the new RDF update. Indeed, RDF-star provides a concise and intuitive method to add information about other triples.

Khayatbashi et al. [50] study on a larger scale the different mappings described by Hartig and benchmark them, but they never consider using different modelings for different elements of the PG 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 property problem, Das et al. study how to use already existing reification techniques to represent properties [10]: 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) [14], an ontology to describe PGs in RDF. As this solution forces the produced data to use the PGO ontology, to the exclusion of any other, it only captures the structure of the PG, and it fails to capture its underlying semantics. Thanks to the Neosemantics[6] 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 generate asserted triples for all PG edges, including those annotated with for example a probability or that are time restricted: for example, if an edge with the label "marriedTo" exists between nodes "Alice" and "Bob", with a property "ended: 2017", the RDF triple `:Alice :marriedto :Bob` should probably not be produced.

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

Gremlinator [51] allows users to query a PG and an RDF database by using the SPARQL language. This is a first step towards federated queries across PGs and RDF graphs. 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. With Expressive Reasoning Graph Store, Neelam et al. [52] propose to store RDF graphs in a JanusGraph PG database. They describe both how to convert RDF data into PG data and SPARQL queries into Gremlin traversals. They show that their approach is faster than Gremlinator and have similar performance as RDF graphs databases.

Excluding the papers on RDF-star, all previously cited work do not propose a user configured conversion: instead, each tool has its own hard coded method to translate PGs into RDF.

The solutions to convert PGs into RDF graph in this thesis will rely on a user defined mapping, named a context

In this regard, Fathy et al. note that sometimes, manually writing a mapping may be time-consuming. They proposed ProGoMap [53], an engine that first generates a putative ontology for the terms in a PG, aligns this ontology with a user chosen pre-existing ontology, and finally converts the PG to an RDF graph with an RML [19] mapping generated from the alignment. The RML mapping generated by this work uses xR2RML [54] to query the PG using the Cypher queries. However, by generating a putative ontology, this work also assumes that all PG edges should be represented by using the same structure in the generated RDF graph. However, in this thesis, we assume that in an idiomatic RDF graph, PG edges that model a "know" relationship and PG edges that model an employer-employee relationship should not be modelled in the same way.

From RDF to PGs Abuoda et al. [55] study the different RDF-star to PG approaches and identified two classes: the RDF-topology preserving transformation which converts each term (including the literals) into a PG node, and the PG transformation that converts literals into property values. They also evaluate the performance of these different approaches. In the former, the PG reflects only the structure of the RDF graph, rather than the domain knowledge. The topology preserving transformations is close to what we referred as syntactic interoperability in the introduction, while PG transformation is closer to semantic interoperability.

In [12], 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 [13], Atemezing and Hyunh propose to use a mapping similar to the former to publish and explore RDF data with a PG tool, namely Neo4j. However, these works offer little customization for the user.

With G2GML [56], 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 proposes a formal definition of PG schemas [57]. 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 effective schema may be extracted from the data, as proposed by Bonifati et al. [58] or Beereen [59]. The idea of using schema as the base for designing a mapping language, in a simplified version from the previously mentioned works, will be the starting point of PRSC in Chapter 5

2.3 A general purpose RDF auto-completion tool

In the work presented in this thesis, the user will be required to drive the conversion from PG to RDF by writing a configuration file. The configuration file will be expressed in RDF, more precisely in the Turtle format. Therefore, the question of how easy it is to write the configuration file arose, and how to help the user writing them.

2.3.1 Where do RDF Triples come from?

RDF data are usually not written by hand. Most approaches to generate RDF data either rely on some kind of abstraction, converting existing data or directly writing programs that output RDF data.

From user input To generate RDF data from user input, the most popular abstractions are forms. Protégé [60] requires the user to fill forms to build their ontology and then generate the corresponding RDF file. The SHACL specification explicitly mentions the possibility to generate forms from property shapes, which has been implemented by systems like Schimatos [61]. Form generators have also been developed for ShEx, the other main shape language for RDF.

Converting data to RDF As described in Section 2.1.2, using tools like RML [19] or JSON-LD [20] is a popular way to produce RDF data from existing data.

Writing programs that output RDF data Software can also directly produce RDF data as an output.

Even by using these approaches, users may still have to write RDF documents by themselves: for example the R2RML mappings must be described in RDF, users may want to fine tune the ontology produced by Protégé...

2.3.2 How do current editors help users?

Plugins for popular code editors have been developed, like *LNKD. tech Editor* [7] and *RDF and SPARQL* [8] for the JetBrain suite. Following the Server Language Protocol [62], a language server for Turtle has been developed by Stardog Union[9]. But all these plugins mainly focus on syntactic checking and coloration.

In [63], Rafes et al. list some of the expected features from a SPARQL auto-completion module. They identify 3 major categories: suggestion of snippets, prefix declaration and auto-completion for Internationalized Resource Identifiers (IRIs). Snippets suggestion is described as being mostly requested by experienced users and can be seen as the step after suggesting terms. Prefix auto-completion is deployed by most editors, through the use of the prefix.cc API[10].

To the best of our knowledge, IRI suggestions in all RDF document editors, like *RDF and SPARQL* and *Yasgui* [64] are limited to proposing all the terms that exist in a given ontology.

[7] https://plugins.jetbrains.com/plugin/12802-lnkd-tech-editor
[8] https://sharedvocabs.com/products/rdfandsparql/
[9] https://marketplace.visualstudio.com/items?itemName=stardog-union.vscode-langserver-turtle
[10] https://prefix.cc

Yasgui filters the list of suggestions depending on the position: for example if the current term is a predicate, all properties in the ontology are displayed and other terms are discarded. This approach is best suited for small ontologies, but for big ontologies, like schema.org[11], the number of suggestions can reach hundreds, making it impractical for users.

Some SPARQL editors like the Flint SPARQL Editor[12] or the one presented by Gombos and Kiss in [65] use intermediate SPARQL queries to help users write their queries. Sparqlis [66] also uses this approach but goes further by exposing an interface in natural language, removing the need for the end-user to know SPARQL. In [67], De la Parra and Hogan first compute the relationships between all types and predicates in the graph, and use the result of this computation to provide auto-completion when the user builds their SPARQL query. All these approaches resort on using the actual data to produce the effective schema of the graph. But in our case, as we are interested in writing new data, these kinds of approaches are not applicable. Hence instead of using the effective schema of the graph we will rely on the expected schema as specified by an RDFS ontology or a set of SHACL shapes.

To help people querying RDF data, including those who are not familiar with SPARQL, tools to abstract queries have been developed. The Sparnatural [68] lets user compose their query by using a visual editor. The visual editor shows the different options and guide the user by listing the possible options. Sparnatural engine uses OWL ontologies as a configuration files, with added annotation properties to choose which classes to use and how to display them. SPARE-LNC [69] is a tool that requires the user to describe their query in a grammar close to natural language, then translates it to proper SPARQL and executes it. However, the controlled natural language supported by SPARE-LNC and its use case is very constrained: the input must follow a precise formal grammar that only supports querying a specific kind of data (modelled traces).

2.3.3 What could be used?

Of the three categories mentioned by Rafes et al. in [63], IRI suggestion is the least studied domain to help users writing RDF document. Yet, the idea behind IRI suggestion is very simple: for a given resource, it would consist in suggesting the predicates related to its types.

RDF documents that describe the predicates that connect different types already exist and are widely spread through ontologies, namely RDFS and OWL mentioned earlier. More recently, the need to validate RDF document lead to the development of the schema languages SHACL and ShEx. These two kinds of schemas, ontologies and validating schemas, could be used to provide IRI suggestion, which will be discussed in Chapter 6.

[11] https://www.schema.org
[12] http://fr.dbpedia.org/sparqlEditor

Chapter 3

PREC: the general framework

In this chapter, we present the common framework, PREC (Property Graph to RDF graph Experimental Converter), for the two PG to RDF converters presented in this thesis: the PREC-C (PREC-Context) converter presented in Chapter 4 and the PRSC (Property Graph to RDF Graph Schema-based Converter) converter presented in Chapter 5.

First, in Sections 3.1 and 3.2, the definition of Angles for PG is presented and discussed with respect to existing PG implementations. The discussion ends with a proposal for a new formal definition of PGs in Sections 3.3 and 3.4.

Then, in Section 3.5, we first present the formal definition of RDF-star graphs, *i.e.* RDF graphs that allow nested triples. Then, we introduce our concept of template triples: triples that are provided by the user to guide the conversion process.

Finally, in Section 3.6, the terminology around the PREC system is presented in addition to the first step towards the proposed PG to RDF conversions.

3.1 Formal definitions of Property Graphs

Definition 1 is an adaptation of Angle's original definition from [21]. In fact, while Angles' formalization of PGs is largely reused in the literature, there are many minor differences in the way different authors define it.

Definition 1 [Property Graphs as defined by Angles in [21]]
A property graph pg is defined as the tuple $(N_{pg}, E_{pg}, src_{pg}, dest_{pg}, labels_{pg}, properties_{pg})$, where:

- N_{pg} and E_{pg} are finite sets with $N_{pg} \cap E_{pg} = \emptyset$. N_{pg} and E_{pg} are respectively the set of nodes and the set of edges of the property graph pg.
- $src_{pg} : E_{pg} \rightarrow N_{pg}$ 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. Considering a property is a key-value pair, it expects two inputs: a node or an edge, and a property key. The output is the property value.

where:

- *Str* denotes the set of all strings.
- V denotes the set of property values. Usually, this set is a super-set of the set of real numbers $\mathcal{R}$ and a super-set of the set of strings *Str*.

The set of all PGs that follow this definition is denoted *APG* (Angles' Property Graphs).

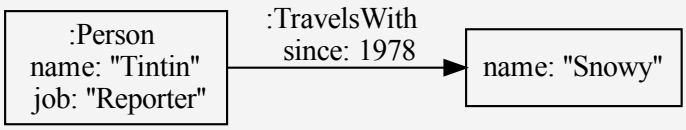


Figure 3.1: A small PG about Tintin that serves as a running example in this chapter

Example 1
The PG exposed on Figure 3.1 can formally be defined as the APG (Angles Property Graph) denoted TT with

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

For a given PG, its nodes and edges are grouped under the term of **element** (or PG element).

Remark 1 [Minor differences between PG formalization inspired by Angles]
Most authors use slightly different formalisms. For example, some authors merge the *src* and the *dest* functions into one function that maps all edges to a pair of nodes. Some others define a set for the set of all labels and do not specify it further. It is the case of the original work of Angles, as it defines respectively a set L and a set P for the set of labels and the set of property keys. Others split the *labels* function to have one that maps the nodes to their set of labels and another one that maps the edges to their set of labels.

However, these differences are minor and do not add or remove expressivity to the model.

Remark 2 [PGs are lazily defined as tuples in this thesis]
Other works always explicitly define any new PG as a tuple
$(N, E, src, dest, labels, properties)$, and define subsequent PGs if needed with a tuple of new symbols.

We find it more convenient and readable to consider that N, E, *src*, *dest*, *labels* and *properties* are implicitly defined for any PG and disambiguated by using an indexed nota-

tion. For example, given a PG x, N_x is its set of nodes, E_x is its set of edges... without the need to explicitly define x as the tuple $(N_x, E_x, src_x, dest_x, labels_x, properties_x)$.

3.2 The need for another PG definition

Despite the PG formal definition of Angles being the consensual definition, this definition is unable to capture all possible implementations.

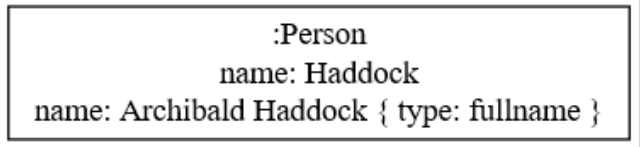


Figure 3.2: A PG whose properties may not be supported by all PG models

Consider the PG exposed in Figure 3.2, composed of a single node. Listing 3.1 shows how to create this node with the Gremlin API:

Listing 3.1: Multiple properties with the same key in Gremlin

```
1  // Create an empty graph
2  gremlin> graph = TinkerGraph.open(); g = traversal().withEmbedded(graph)
3  ==>graphtraversalsource[tinkergraph[vertices:0 edges:0], standard]
4
5  // Add a node with the person label
6  gremlin> g.addV("person")
7  // With a property whose key is name and whose value is "Haddock"
8                                  .property("name", "Haddock")
9  // With another property, added in the list of properties,
10 // whose value is name and whose value is "Archibald Haddock"
11                                 .property(list, "name", "Archibald Haddock",
12 // Add to this property the meta property whose key is "type"
13 // and whose value is "fullname"
14                                   "type", "fullname"
15                         )
16 ==>v[0]
17
18 // Only one node in the graph
19 gremlin> g.V()
20 ==>v[0]
21
22 // Its label is person
23 gremlin> g.V().label()
24 ==>person
25
26 // It has two properties with the same key (unsupported by Angles definition)
27 gremlin> g.V()[0].properties()
28 ==>vp[name->Haddock]
29 ==>vp[name->Archibald Haddock]
30
31 // The "name->Haddock" property has no meta properties
32 gremlin> g.V().properties("name").hasValue("Haddock").properties()
33
34 // The "name->Archibald Haddock" property has one meta property
35 // - Not natively supported by Angles definition
36 // - Could be emulated by integrating meta properties in the set V
37 gremlin> g.V().properties("name").hasValue("Archibald Haddock").properties()
38 ==>p[type->fullname]
```

- Line 2 creates the PG.
- In Lines 5–16, the node is created:
  - Line 6 creates a new vertex whose label is Person.

  - Line 8 adds a property with the key “name” and the value “Haddock” to the vertex created in line 6.
  - Line 11 adds another property with the key “name” and the value “Archibald Haddock”. Note that because the insertion strategy is set to “list”, the property is added without any consideration about other properties that already exist.
  - Line 14 adds a meta property to the property added by line 11: a property whose key is “type” and whose value is “fullname”.
- Lines starting from line 18 are dedicated to printing the created PG to check if it is correct, in particular:
  - In lines 27–29, both properties with the same key exist.
  - In lines 31–32, the first property has no meta properties.
  - In lines 34–38, the second property has the requested meta property.

This example illustrates that at least one property graph engine, the TinkerGraph engine supports features that cannot be represented with Angles definition. Indeed, in Angles's definition, the *properties* function is supposed to match a pair of a PG element and a key with a value. This definition does not fit the example exposed in Listing 3.1 in two ways:

- It is unable to represent the fact that a property key can be used multiple times, like the “name” properties in our example. However, this is an explicit feature of Gremlin, through its `Cardinality` enum that can be passed to the `property` function to define the insertion strategy in case a property with the given key already exists: `Cardinality.single` sets the key to the value, `Cardinality.set` only inserts a new property if the key-value pair does not exist and `Cardinality.list` always adds a new property. In Angles's formalism, only the `Cardinality.single` behavior is supported.
- The fact that the second property has a property, *i.e.* a meta property, is not explicitly supported.

Following the Gremlin API, it may be possible for an implementation to use edges and properties as the source or the destination of another edge, like demonstrated in Listing 3.2. However, to the best of our knowledge, no property graph engine actually implement this behavior, for example in the same listing, an exception is thrown because it is not supported, and this possibility is not considered in the rest of the thesis.

Listing 3.2: TinkerGraph does not support meta edges

```
// Creating a graph with three nodes: source, destination and metadestination
gremlin> graph = TinkerGraph.open(); g = traversal().withEmbedded(graph)
==>graphtraversalsource[tinkergraph[vertices:0 edges:0], standard]
gremlin> g.addV("source")
==>v[0]
gremlin> g.addV("destination")
==>v[1]
gremlin> g.addV("metadestination")
==>v[2]

// Creating source->destination
gremlin> g.V().hasLabel("source").as('a')
           .V().hasLabel("destination").as('b')
           .addE("edge").from('a').to('b')
==>e[3][0-edge->1]

// Testing the edge
gremlin> g.V().hasLabel("source").out().label()
==>destination

// Creating (source->destination)->metadestination
```

```
gremlin> g.E().hasLabel("edge").as('a')
          .V().hasLabel("metadestination").as('b')
          .addE("metaedge").from('a').to('b')

// As of 3.4.13, an exception is thrown by TinkerGraph
org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerEdge cannot be cast to
org.apache.tinkerpop.gremlin.structure.Vertex
Type ':help' or ':h' for help.
```

3.3 Gremlinable Property Graphs

To be able to fully support the possibilities given by the Gremlin API, we propose a larger definition of PGs.

Definition 2 [Formal definition of a Gremlinable PG]
A *Gremlin-able Property Graph* (GPG) pg is a PG defined as follows:

- N_{pg} is the finite set of nodes of the PG pg.
- E_{pg} is the finite set of edges of the PG pg.
- P_{pg} is the finite set of properties of the PG pg.
- $src_{pg} : E_{pg} \to N_{pg}$ is a total function. It maps each edge to its source node.
- $dest_{pg} : E_{pg} \to N_{pg}$ is a total function. It maps each edge to its destination node.
- $labels_{pg} : N_{pg} \cup E_{pg} \to 2^{Str}$ is a total function. It maps each node and edge to its finite set of labels.
- $properties_{pg} \subset (N_{pg} \cup E_{pg} \cup P_{pg}) \times P_{pg}$ is a relation between each node, edge or property and its properties.
- $propdetails_{pg} : P_{pg} \to String \times V$ is a total function. It maps each property to a pair containing its key and its value.

These sets must comply with the following constraints:

- N_{pg}, E_{pg} and P_{pg} are pairwise disjoint.
- The relation $properties_{pg}$ forms a directed forest whose roots are elements (*i.e.* nodes or edges):
  - Each property has exactly one parent: $\forall p \in P_{pg}, \exists! m, m\ properties_{pg}\ p$
  - Each property has exactly one PG element (node or edge) as an ancestor: $\forall p \in P_{pg}, \exists n \in \mathbb{N}, \exists! m \in N_{pg} \cup E_{pg}, m\ (properties_{pg})^n\ p$
  - From the two previous points, it follows that there are no cycles in the relation $properties_{pg}$.

The set of all GPGs is denoted $GPGs$.

This new definition covers the cases missing in Angles's definition but covered by the features described in Section 3.2:

- Meta-properties are handled by the fact that the *properties* relation allows properties, the elements of the set P, as its first member *i.e.* as the holder of the property.
- Multiple properties sharing the same property key are handled by the fact that the *properties* function does not use the property key as a part of its domain anymore. Instead, it is a property that relates the holder with the held property, and the *propdetails*

function maps each property to the property key-value pair. It allows multiple properties to use the same property key[1].

Definition 3 [NEP and kind]
A *NEP* (node or edge or property) is something that is either a PG node, a PG edge or a property. In other words, it is either a PG element or a property.

The *kind* of a NEP is "node" if the NEP is a node, "edge" if the NEP is an edge and "property" if the NEP is a property.

The $kind_{pg}$ function is defined formally as:

$$kind_{pg}(m) = \begin{cases} \text{“}node\text{”} & \text{if } m \in N_{pg} \\ \text{“}edge\text{”} & \text{if } m \in E_{pg} \\ \text{“}property\text{”} & \text{if } m \in P_{pg} \end{cases}$$

Notation: In the rest of this thesis, the set of the NEPs of a PG, $N_{pg} \cup E_{pg} \cup P_{pg}$, may be written as NEP_{pg}.

Example 2 [Formalization of PG in Figure 3.1 with the GPG formalism]
The PG in Figure 3.1 can be formalized under the GPG definition as the PG TT such that:

- $N_{TT} = \{n_1, n_2\}\,; E_{TT'} = \{e_1\}\,; P_{TT'} = \{p_1, p_2, p_3, p_4\}$
- $src_{TT} = \{e_1 \mapsto n_1\}\,; dest_{TT} = \{e_1 \mapsto n_2\}$
- $labels_{TT} = \{n_1 \mapsto \{\text{“}Person\text{”}\}\,; n_2 \mapsto \emptyset; e_1 \mapsto \{\text{“}TravelsWith\text{”}\}\}$
- $properties_{TT} = \{(n_1, p_1), (n_1, p_2), (n_2, p_3), (e_1, p_4)\}$
- $propdetails_{TT} = \begin{Bmatrix} p_1 \mapsto (\text{“}name\text{”}, \text{“}Tintin\text{”}); p_2 \mapsto (\text{“}job\text{”}, \text{“}Reporter\text{”}) \\ p_3 \mapsto (\text{“}name\text{”}, \text{“}Snowy\text{”}); p_4 \mapsto (\text{“}since\text{”}, 1978) \end{Bmatrix}$

Example 3 [One possible formalization of the PG in Figure 3.2]
The PG in Figure 3.2 can be formalized under the GPG definition as the PG Hd such that:

- $N_{Hd} = \{n_1\}$
- $E_{Hd} = \emptyset$
- $P_{Hd} = \{p_1, p_2, p_3\}$
- $src_{Hd} = dest_{Hd} = \emptyset \to \emptyset$
- $labels_{Hd} = \{n_1 \mapsto \{\text{“}Person\text{”}\}\}$
- $properties_{Hd} = \{(n_1, p_1), (n_1, p_2), (p_2, p_3)\}$
- $propdetails_{Hd} = \begin{Bmatrix} p_1 \mapsto (\text{“}name\text{”}, \text{“}Haddock\text{”}), \\ p_2 \mapsto (\text{“}name\text{”}, \text{“}ArchibaldHaddock\text{”}), \\ p_3 \mapsto (\text{“}type\text{”}, \text{“}fullname\text{”}) \end{Bmatrix}$

The PG Hd can not be expressed following Angles's definition.

[1]By consequence, the property key is not strictly speaking a key anymore. However, to remain consistent across the whole thesis, the term "property key" is used in all cases.

Remark 3 [From Gremlinable PGs to Angles PGs]
Consider a GPG satisfying the following hypotheses:

(a) It contains no meta properties.

(b) No element has two properties with the same key.

We show below that all such GPGs can be expressed with Angles's definition, therefore these hypotheses are named "Angles' hypotheses".

Proof. Note that it is always possible to rewrite the relation $properties_{pg} \subseteq NEP \times P_{pg}$ into a function $properties_{pg} : NEP \to 2^{P_{pg}}$ that maps each NEP to the set of properties that it is in relation with.

Let pg be a GPG that follows Angles's hypotheses. We have:

- N_{pg}, E_{pg} and P_{pg} are pairwise disjoint finite sets
- $src_{pg} : E_{pg} \to N_{pg}$ is a total function
- $dest_{pg} : E_{pg} \to N_{pg}$ is a total function
- $labels_{pg} : N_{pg} \cup E_{pg} \to 2^{Str}$ is a total function
- $properties_{pg} : N_{pg} \cup E_{pg} \cup P_{pg} \to 2^{P_{pg}}$ is a total function
- $propdetails_{pg} : P_{pg} \to Str \times V$ is a total function

As pg contains no meta property, the domain of $properties_{pg}$ can be simplified by removing P_{pg}: $properties_{pg} : N_{pg} \cup E_{pg} \to 2^{P_{pg}}$ (Hypothesis (a))

The set P_{pg} only appears twice: once in the image of $properties_{pg}$ and once as the domain of $propdetails_{pg}$ which is a total function.

We can remove P_{pg} by replacing any occurrence of a member of P_{pg} with its image through $propdetails_{pg}$. The $propdetails_{pg}$ function can be removed, and the *properties* function is rewritten as follows: $properties_{pg} : N_{pg} \cup E_{pg} \to 2^{Str \times V}$.

We can rephrase (b) as $\forall m \in N_{pg} \cup E_{pg}, properties_{pg}(m)$ is a function, and apply the following transformation on $properties_{pg}$:

$$
\begin{array}{lll}
properties_{pg} : N_{pg} \cup E_{pg} \to 2^{String \times V} & \text{(Total function)} & \\
\Leftrightarrow properties_{pg} : N_{pg} \cup E_{pg} \to (String \to V) & \text{(Total function)} & \text{(b)} \\
\Rightarrow properties_{pg} : (N_{pg} \cup E_{pg}) \times String \to V & \text{(Partial function)} & \text{[Uncurryfication]}
\end{array}
$$

We end up with the following PG formal definition:

- N_{pg} and E_{pg} are disjoint finite sets.
- $src_{pg} : E_{pg} \to N_{pg}$ is a total function.
- $dest_{pg} : E_{pg} \to N_{pg}$ is a total function.
- $labels_{pg} : N_{pg} \cup E_{pg} \to 2^{Str}$ is a total function.
- $properties_{pg} : (N_{pg} \cup E_{pg}) \times Str \to V$ is a partial function.

Conversely, all PGs that are covered by Angles' definition are trivially covered by the GPG definition: Given a PG h that can be expressed following Angle's definition, build the set P_h and a new $properties'_h$ function such as $|P_h| = |Dom(properties_h)|$ and

$$
\begin{array}{r}
\forall((holder, key), value) \in properties_h, \exists p \in P_h, \\
(holder, p) \in properties'_h \\
\wedge propdetails_h(p) = (key, value)
\end{array}
$$

where $properties'_h$ is the $properties_h$ definition in the GPG formalism.

Hence, following the hypothesis of Angles, our definition can be simplified to the one proposed by Angles. Without these restrictions, the GPG definition encompasses more PG implementations, *i.e.* Gremlin-compliant PGs like the PG in Figure 3.2.

□

Note that the PG TT from Example 2 is not the only one that can be represented by Figure 3.1: as the identity of the nodes, edges and properties are not specified on the Figure, any arbitrary element could have been used in place of the one chosen in TT. In other words, Figure 3.1 does not show a single PG, but an entire class of isomorphic PGs.

Definition 4 [Renaming function]
For all sets $N_1, N_2, E_1, E_2, P_1, P_2$ where $N_1 \cap E_1 \cap P_1 = \emptyset$ and $N_2 \cap E_2 \cap P_2 = \emptyset$, a renaming is a bijective function $\phi : N_1 \cup E_1 \cup P_1 \to N_2 \cup E_2 \cup P_2$ where $\forall n \in N_1, \phi(n) \in N_2 \wedge \forall e \in E_1, \phi(e) \in E_2 \wedge \forall p \in P_1, \phi(p) \in P_2$.

Example 4
Let TT' be another formalization of the PG in Figure 3.1 that uses the sets $N_{TT'} = \{a, b\}$, $E_{TT'} = \{c\}$ and $P_{TT'} = \{d, e, f, g\}$.

An example of a renaming function ϕ_{TT} from $N_{TT} = \{n_1, n_2\} \cup E_{TT} = \{e_1\} \cup P_{TT} = \{p_1, p_2, p_3, p_4\}$ to $N_{TT'} = \{a, b\} \cup E_{TT'} = \{c\} \cup P_{TT'} = \{d, e, f, g\}$ is
$\phi_{TT} = \{n_1 \mapsto a; n_2 \mapsto b; e_1 \mapsto c; p_1 \mapsto d; p_2 \mapsto e; p_3 \mapsto f; p_4 \mapsto g\}$.

Definition 5 [Property Graph renaming]
Let pg and pg' be two GPGs and ϕ be a renaming function from NEP_{pg} to $NEP_{pg'}$. $pg' = \phi(pg)$ is defined as follows:

- $N_{pg'} = \{\phi(n) \mid n \in N_{pg}\}$
- $E_{pg'} = \{\phi(e) \mid e \in E_{pg}\}$
- $src_{pg'} = \{e \mapsto \phi(src_{pg}(\phi^{-1}(e))) \mid e \in E_{pg'}\}$
- $dest_{pg'} = \{e \mapsto \phi(dest_{pg}(\phi^{-1}(e))) \mid e \in E_{pg'}\}$
- $labels_{pg'} = \{m \mapsto labels_{pg}(\phi^{-1}(m)) \mid m \in N_{pg'} \cup E_{pg'}\}$
- $properties_{pg'} = \{(\phi^{-1}(m), \phi^{-1}(p)) \mid (m, p) \in properties_{pg}\}$
- $propdetails_{pg'} = \{m \mapsto (key, value)) \mid (key, value) = propdetails_{pg}(\phi^{-1}(m))\}$

Example 5
Let us consider back TT, the PG about Tintin defined in Example 1, TT' the other formalization of the same PG and ϕ_{TT} the renaming function introduced in the Example 4.

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

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

Definition 6 [Isomorphic Property Graph]
Two PGs pg and pg' are isomorphic iff there exists a renaming function ϕ such that $rename(\phi, pg) = pg'$.

Note that both TT and TT' from Examples 1 and 4 match the graphical representation given in Figure 3.1. An informal way to define the isomorphism between two PGs is to check if they have the same graphical representation.

Existing works [70, 71] on PG query languages focus on extracting the properties of some nodes and edges, and never look for the exact identity of NEPs. 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 as a practical matter.

Remark 4
The formal definition of the PG isomorphism has been built on the GPG definition provided in Definition 2. From Remark 3, we can deduce that the isomorphism between two PGs defined from Angles definition is defined as two graphs for which there is a renaming from the set of nodes and edges from one PG to the other.

3.4 Discussion about Gremlinable Property Graphs

In the previous section, we introduced a new formalization of Property Graphs: the Gremlinable Property Graphs formalized by Definition 2. As pointed out by Remark 3, this formalization can model all PGs that can be modeled following the state-of-the-art definition of the PGs, the definition from Angles, recalled by Definition 1.

It is also able to support a super-set of the PGs that are supported by Neo4j and the TinkerPop implementation that we are aware of: both Neo4j and the Gremlin API are unable to support multiple labels on the same edge, despite Angles' formalization supporting this feature.

In this thesis, the Gremlinable Property Graph definition serves as a starting point, as it is able to support both the state-of-the-art standard definition and the existing implementations.

When converting Property Graphs, in this thesis, depending on the expressivity of the studied transformation method, the set of convertible PGs may be restricted. Furthermore, for simplicity, as long as the conversion from the Gremlinable PG definition to another PG definition is computable, we allow ourselves to use the later PG definition.

Table 3.1: List of prefixes used in this thesis

Prefix	IRI
rdf	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
xsd	`http://www.w3.org/2001/XMLSchema#`
ex	`http://example.org/`
foaf	`http://xmlns.com/foaf/0.1/`
schema	`http://schema.org/`
prec	`http://bruy.at/prec#`
pvar	`http://bruy.at/prec-var#`

For example, in Chapter 5, the proposed conversion only supports PG without meta properties *i.e.* the subset of GPGs that are covered by Angles' original definition (Definition 1), as we have shown in Remark 3. In that chapter, we therefore work with Angles' definition, as it is better suited for this use case. In Chapter 4, as PREC-C supports features not supported by Angles' definition, for instance meta properties, the used formalism is the Gremlinable PG.

3.5 Formal definitions of RDF and template graphs

In this section, first we recall the formal definition of RDF graphs, with the addition of RDF-star. Then, we introduce the concept of template triples.

3.5.1 RDF(-star) graphs

Definition 7 [Atomic RDF terms]
Let I be the infinite set of IRIs, $L = Str \times I$ be the set of literals and B be the infinite set of blank nodes. The sets I, L and B are disjoint.

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

Notation: In the examples, the IRIs, the elements of I, will be either noted as full IRIs between brackets, *e.g.* `<http://example.org/Tintin>` or by using prefixes to shorten the IRI *e.g.* `ex:Tintin`. The list of prefixes used in this paper is described in Table 3.1.

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

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

Definition 8 [RDF(-star] triples and graphs)
The set of all RDF triples 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.

Both the atomic RDF terms defined in Definition 7 and RDF triples are *terms*. A triple used in another triple, in subject or object position, is a *quoted triple*.

Remark 5 [RDF-star is conflated with the vanilla RDF model]
For the sake of readability, although RDF-star is not yet part of the official RDF recommendation [72], we conflate RDF-star and RDF in this thesis. When we mention an RDF triple or an RDF graph, we allow them to contain quoted triples.

Example 6

1. The triple $(ex{:}tintin, rdf{:}type, ex{:}Person)$ is an element of *RdfTriples*. Its Turtle representation is `ex:tintin rdf:type ex:Person`.
2. The RDF graph exposed in Listing 5.1 is composed of 5 triples written in Turtle format. In our formalism, the second triple, `_:tintin foaf:name "Tintin"`, is $(\_{:}tintin, foaf{:}name, \text{“}Tintin\text{”}^{xsd:string})$.
3. $(ex{:}tintin, ex{:}travelsWith, ex{:}snowy)$ is an element of *RdfTriples*.
   $((ex{:}tintin, ex{:}travelsWith, ex{:}snowy), ex{:}since, \text{“}1978\text{”}^{xsd:integer})$ is an element of *RdfTriples* that has a quoted triple in subject position.

Definition 9 [Term membership]
The $\in$ operator is extended to triples to check if a term is part of a triple.

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

Example 7 [Term membership examples]

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

Definition 10 [List of blank nodes used in an RDF graph]
For every RDF graphs rdf, $BNodes(rdf)$ is the list of blank nodes in rdf *i.e.* $\forall rdf \subseteq RdfTriples, BNodes(rdf) = \{bn \in B \mid \exists t \in rdf, bn \in t\}$.

Example 8
Let GTT be the RDF graph exposed on Listing 5.1. $BNodes(GTT) = \{\_{:}tintin, \_{:}snowy\}$

3.5.2 Template graphs

PREC uses a novel templating system. Already existing systems, like RML [73][19], use a complex system to express which triples to produce. Some other works like SPARQL generate [74] are not expressed in terms of pure RDF documents but through SPARQL queries.

However, RDF-star opens new possibilities in terms of templating. Instead of using convoluted constructions to describe how to build triples, it is now possible to just use the triples themselves[2]. The only remaining issue is how to describe the terms that are going to change, *i.e.* the terms that depend on the data.

The exact method of how placeholders are replaced by RDF terms depends on the conversion method, *i.e.* PREC-C in Chapter 4 and PRSC in Chapter 5 will use different methods to produce RDF triples from template triples.

Definition 11 [Term placeholders]
Placeholders are special terms that can only be used in template triples (they can not be used in RDF triples). They are not included in any of the previously defined sets.

There are two types of placeholders:

- *Node placeholders* can be used in the subject and in the object position of a template triple. They are bound to be replaced with blank nodes or IRIs. The set of node placeholders is PN.
- *IRI placeholders* can be used in any position of a template triple. they are bound to be replaced with IRIs. The set of IRI placeholders is PI.
- *Literal placeholders* can only be used in the object position of a template triple. They are bound to be replaced with literals. The set of literal placeholders is PL.

The exact composition of the sets PN, PI and PL depends on the conversion method. The list of their members will be described at the beginning of the description of each converter.

Multiple strategies can be applied to define the placeholders:

- Each algorithm define a finite list of fixed placeholders, prefixed by ? in the formal definitions. For example, $?self$ is a placeholder that is bounded to be replaced with the NEP itself. This strategy is mostly used to define the placeholders in the sets PN and PI. In the implementation, these placeholders are represented by IRIs in the reserved namespace `pvar`, *e.g.* $?self$ is represented by `pvar:self`.

[2]Named graphs could also have been used instead of a list of quoted triples to list the triples to produce. However, to the best of our knowledge, there are no work discussing this possibility.

- For literal placeholders, another possible strategy is to use infinite sets. In the PRSC conversion presented in Chapter 5, the sets of literal placeholders PL will be defined as the set of all pairs in the set $Str \times \{valueOf\}$, for example $(name, valueOf)$. In the implementation, this is reflected by the introduction of the special datatype `prec:valueOf`, introducing placeholders like `"name"^^prec:valueOf`. These placeholders will be replaced by the algorithm by the value of the property that is named by the placeholder, for instance the value of the property whose key is "name".
- Other strategies are possible, but are not applied in this thesis. For example, the implementation could use placeholders such as `"http://example.org/{name}"^^prec:templatedIRI` to produce IRIs depending on the content of the PG. However, note that using placeholders that are implemented as literals with a special datatype restrict the usage of these literals to the object position in the standard RDF-star model. Using these terms in all positions requires to use the extended RDF model that allows any kind of term to appear in all positions. However, when these use cases are met, a more sophisticated templating system like the one from R2RML may be more relevant, as they already support these features at the cost of being more verbose.

Definition 12 [Template triples]
The set *Templates* of all *template triples* is defined inductively as follows:

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

Any subset of *Templates* is named a **template graph**. Note that unlike *RdfTriples*, the elements of *Templates* can not contain blank nodes but can contain placeholders. Also note that the set of template triples are formally different from the set of template triples.

Similarly to Definition 8, template triples can contain nested template triples. Similarly to Definition 9, the membership operator $\in$ is extended to template triples.

Example 9
Consider that ?*self* is a member of the set PN.

The template triple $(?self, rdf{:}type, ex{:}Person)$ is a member of *Templates* but not of *RdfTriples* because it uses an element of PN.

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

Note that the structure of the template triple with ?*self* and the structure of the second triple is very similar: A converter is expected to produce triples like the second triple from the template triple by replacing the placeholder ?*self* with appropriate terms.

3.6 PREC (PG to RDF graph Experimental Converter)

3.6.1 The terminology around PREC

Figure 3.3 shows the main component in the PREC framework, both as formally defined in this thesis on the left hand-side and as implemented on the right-hand side.

In both versions, the input is composed of the PG to convert and a context, and the output is an RDF graph. A context is the input provided by the user to guide the transformation of the PG. In the formal definitions, it is a function that tells how to map the elements of the PG to template graphs. In the implementation, it takes the form of a configuration file, expressed in RDF.

As mentioned previously, two converters are presented in this thesis: the *PREC-C* converter that is presented in Chapter 4 and the *PRSC* converter presented in Chapter 5. The contexts that are used for PREC-C and for PRSC are very different: there is no ambiguity if one

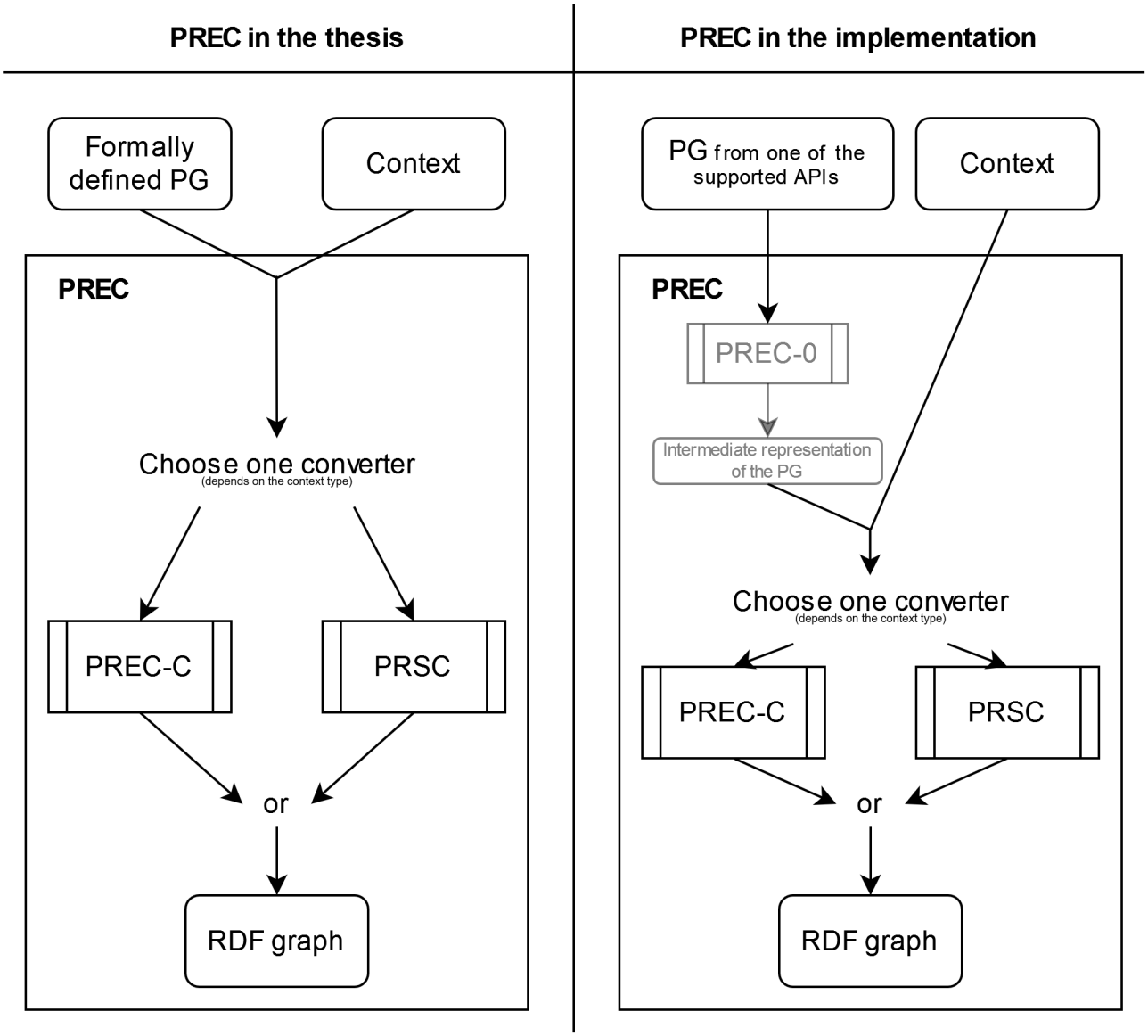


Figure 3.3: The main components of PREC, both as formally defined and in the implementation

converter is used or the other from the content of the context.

Formally, these converters do no share any step: their computations are defined separately. However, they still share some common concepts, like the concept of template graphs presented in Section 3.5.2 or the concept of Blank Node Property Graphs presented in the next section. As the converters share no part during the processing, and as the PREC-C contexts and PRSC contexts are trivially distinguishable, following the formal definitions, PREC only serves as a term to talk about both PREC-C and PRSC.

In the implementation, the PREC-C and the PRSC converter are not directly called by the PREC framework. Because of the great diversity of PG engines and APIs, before the context is applied, the PG is retrieved using one of the supported PG querying API (at the time of writing, Neo4j or Gremlin) and stored in memory. This step is referred as the PREC-0 (PREC with 0 contexts) step. The produced "intermediate representation of the PG" is an RDF graph, but the format is not relevant as this intermediate representation is produced to be used as a PG input by the PREC-C or the PRSC implementation. The benefit of this architecture is to decouple the process of adding a new supported API and the process of adding a new conversion algorithm. When the "intermediate representation of the PG" has been produced, the context is read and the converter corresponding to the context is used. As the input format of the context is the same for both converters, an RDF file, and as the context is expressed as a list of rules to apply, we say that contexts can be written using two distinct rulesets: the PREC-C ruleset and the PRSC ruleset.

In the following two chapters, we start by formally defining the corresponding converters, then present some consideration specific to their respective implementations.

3.6.2 Blank node Property Graphs

First, note that in the PG definitions, the sets N, E and P are very loosely characterized, and their exact members are considered not important. The only important aspects are that they are finite and disjoints. Implementations will often use some identifiers that can be queried by the user: for example, Neo4j use numerical identifiers for the node and edges. However, these identifiers are generally not considered, and graphs are queried using the content of some properties and then by traversing the graphs through the edges. It means that in practice, two isomorphic PGs are considered equal by the users.

Note that the set of blank nodes B in RDF graphs is also loosely characterized: its only distinctive trait is that it is disjoint from the sets of IRIs and literals.

When implementing a PG to RDF converter without any information loss, the first issue is to decide how each NEP will be represented. The easiest method is to assign a blank node to each NEP to identify each one of them.

When formally defining the conversion, we also need to define this step of mapping the NEPs to blank nodes. However, we mentioned that the exact identity of the NEP is not important. 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 pg, we can build an isomorphic PG pg' such that $N_{pg'} \cup E_{pg'} \cup P_{pg'} \subset B$. Being isomorphic to pg, pg' is indistinguishable from pg for any practical purpose, because the exact *identity* of nodes and edges is not important: only the structure and the values of the PG are. Therefore, without any loss of generality, we can restrict our work to PGs whose nodes, edges and properties are elements of B.

Definition 13 [Blank node Property Graph]
A blank node Property Graph is a Property Graph for which all NEPs are blank nodes.

In the following, for any set of Property Graph named X, we will denote its subset of blank node properties by BX:

$$BX = \left\{ pg \in X \;\middle|\; \begin{array}{ll} & N_{pg} \subset B \\ \wedge & E_{pg} \subset B \\ \wedge & P_{pg} \subset B \quad \text{if the set } P_{pg} \text{ exists in this PG formalization} \end{array} \right\}$$

In particular, a Blank node Gremlinable Property Graph (BGPG) is a Gremlinable Property Graph, a member of $GPGs$, whose nodes, edges and properties are blank nodes. The set of all BGPG is noted $BGPG$.

$BGPG = \{g \in GPGs \mid NEP_g \subset B\}$

Remark 6 [All Property Graph sets in this thesis]
The full list of PG sets that are used in this thesis is as follows:

- Gremlinable Property Graphs, denoted by the set GPG, as defined in Definition 2 and its corresponding $BGPG$ set.
- Angles Property Graphs, denoted by set APG, as defined in Definition 1 and its corresponding $BAPG$ set. It is the usual PG description. The $BAPG$ set will be used in Chapter 5.
- PREC-C compatible Property Graphs, denoted by set CPG, will be defined in Definition 14. Its corresponding Blank Node Property Graph set is $BCPG$. CPG is a subset of the BPG property graphs where all edges have one and only one label. The $BCPG$ set will be used in Chapter 4.

Similarly to the usual PGs defined in Definition 1, two BPGs are considered to be distinct and their sets of PG elements are only considered for this given PG. To the best of our knowledge, in all existing works, the possibility of an element e being shared by two PGs is never considered as the only semantic of e is a local semantic in a given PG pg through the src_{pg}, $dest_{pg}$, $labels_{pg}$, $properties_{pg}$, and $propdetails_{pg}$ if it exists in the currently used PG formalization, functions. Moreover, because two isomorphic PGs are considered, if two PGs were to share the same element e, a PG isomorphic to one of them would be instead considered to ensure that the PG elements of two PGs are distinct, and usual works would consider that this step would not lead to any loss of information. Likewise, for two given BPGs, the sets of their PG elements is considered distinct. This is consistent with how the merge operation of RDF graphs is defined [75]; to merge two RDF graphs, the shared blank nodes of the two RDF graphs are first distinguished before performing the mathematical union of the RDF graphs.

The PG to RDF graph conversion functions described in this thesis are only defined for Blank node Property Graphs. In other words, the PG input in Figure 3.3 is always a BPG. If the user does not have a BPG but only a PG, they are simply required to build an isomorphic BPG which is a trivial operation.

This restriction does not impede the generality of our work, and provides several advantages:

- In formal definitions, this enables us to define the PG to RDF conversion as a deterministic operation, *i.e.* a function. It is beneficial, in particular when trying to prove the reversibility of the conversion.

- Implementation-wise, building a BPG isomorphic to the PG to convert is equivalent to assigning to each node, edge and property (if applicable in the case of properties) a blank node, and sticking to this choice for the duration of the conversion process. Therefore, this assignation can be seen as the first step of the conversion. In this thesis, as we will never introduce any further blank node, this step is the only step that is not deterministic. Note that in the RDF world, two graphs which only differ in the name of their blank nodes, *i.e.* two isomorphic RDF graphs, are usually considered equal.

Chapter 4

PREC-C: a low level converter

In this chapter, we introduce the first converter named PREC-C (for PREC context).

The PREC-C algorithm has been designed by using the transformation performed by NeoSemantics as a starting point. NeoSemantics is a Neo4j PG to RDF converter (and back) that produces a triple for each label, a triple for each edge and a triple for each property. The predicate of triples forged from edges and properties, and the object of triples forged from labels are forged from the label or the property key. However, this approach suffers from two issues: 1) it does not allow the user to easily choose to which IRI the label or property key is mapped, and 2) more generally, it does not let the user choose the structure of the produced RDF graph. For example, consider the running example and imagine that Snowy stops travelling with Tintin. In this case, we would want to avoid producing an RDF triple that asserts that Tintin still travels with Snowy. However, NeoSemantics does not allow to not produce this triple.

To tackle this issue, we want to enable the user to specify how to convert into RDF the PG for which they do not want the default representation. To specify which part of the PG for which they want to specify the transformation, we define the concept of selectors. There are three kinds of selectors: node selectors, edge selectors and properties selectors. Each kind of selector selects NEPs using the label or the property key. The PREC-C context is defined as a mapping from selectors to template graphs: after a collection of NEPs has been selected by a selector, the template graph is used to produce RDF triples. As different selectors may be mapped to different template graphs, this approach makes it possible to specify different RDF structures for different parts of the PG. In particular, each template specifies *de facto* the IRI of the corresponding label or property key.

In Section 4.1, we first formally define how PGs are transformed into RDF graphs using the PREC-C context. In particular, the notions of selectors and PREC-C context are formally defined, and we specify how the different template graphs interact with each other. Then in Section 4.2, we propose a method to express PREC-C contexts using RDF graphs through the definition of the PREC-C ontology, and relate the PREC-C ontology with the formal definition.

A PREC-C context consists in overriding a default behavior: similarly to NeoSemantics which is always able to convert any PG (as it does not have the concept of context to begin with), PREC-C is able to produce an RDF graph from a PG and an empty context.

It is worth noting that, in our implementation described on the right-hand side of Figure 3.3, the PREC-0 step is equivalent to applying PREC-C with an empty context. Then the implementation of the PREC-C step transforms the intermediate representation based on the context, this transformation being the identity if the context is empty.

4.1 Formal definition of PREC-C

In this section, the PREC-C algorithm is formally described. It takes as input a PG and a context and outputs an RDF graph. Intuitively, the context allows the user to specify how the different NEPs of the PG are mapped to RDF constructs.

In order to better explain the design choices of the PREC-C algorithm, we will present several versions of it; each adding more features to the previous one.

4.1.1 Characterization of the compatible graphs

The PREC-C conversion algorithm is only defined for a subset of Property Graphs: the one for which all edges have one and only one label. This is motivated by the fact that we consider Cypher and Gremlin as the mainly used APIs for Property Graph querying, and they both share the common restriction of one and only one label for edges.

Definition 14 [PREC-C compatible Property Graphs]
A PREC-C compatible Property Graph (CPG) is a Gremlinable Property Graph for which all edges have one and only one label:

$$CPGs = \{pg \in PGs \mid \forall e \in E_{pg}, |labels_{pg}(e)| = 1\}$$

The set of all CPGs is denoted $CPGs$.

As discussed in Definition 13, the set $BCPGs$ is the subset of PREC-C compatible Property Graphs where all nodes, edges and properties are blank nodes:

$$BCPGs = \{pg \in BPGs \mid \forall e \in E_{pg}, |labels_{pg}(e)| = 1\}$$

4.1.2 First iteration: using a default context

Algorithms 1, 2 and 3 describe the operation performed by PREC-C to convert a PG to an RDF graph, without any consideration about the user's choices. The RDF graph produced by this first version of PREC-C corresponds exactly to the PREC-0 pre-processing used in our implementation as described in Section 3.6.

Overview of the main algorithm *precc* in Algorithm 1 The entry point of the PREC-C conversion is the *precc* function described in Algorithm 1. It takes a PG named *pg* that belongs to the set *BCPGs*.

The *precc* function is composed of three main loops, one per kind of NEP, that loops on each NEP of this kind. The algorithm mainly consists in the following steps, for each NEP:

- Compute the *selectors* matching it:
  - For nodes, there is one independent selector for each label.
  - For edges, there is only one selector based on the label.
  - For properties, there is only one selector based on the property key.
- Pass the selector(s) to the corresponding *p0NodeLabelRules*, *p0EdgeRules* or *p0PropertyRules* function to get a template graph. The function returns a template graph that describes the structure of the triples to produce.
- Apply the *build* function to the template graph to convert the *placeholders*. Then, the produced triples are merged with the output RDF graph.

The *build* function in Algorithm 2 The *build* function relies on a sub-function named β. The β function recurses on the given triple until it is called on an atomic RDF term[1] or a placeholder. Then it checks its value, and if it is a placeholder, it replaces it with a value extracted from the Property Graph.

Note that a utility function *getHolder* is defined to find the NEP holding a given property.

The PREC-0 rules The PREC-0[2] functions, *i.e.* the *p0NodeLabelRules*, the *p0EdgeRules* and the *p0PropertyRules* functions, provide for a given label or property key the corresponding template graph. They are described in Algorithm 3.

These functions suppose that there is a function *forgeIRI* that can build an IRI from an IRI prefix and a string literal.

Note that the *p0NodeLabelRules*, *p0EdgeRules* and *p0PropertyRules* functions shape the schema of the produced RDF graph. The produced RDF graph complies with the schema presented in Figure 4.2. The idea of this representation is to describe the PG with generic RDF constructions, in particular by using the standard RDF reification for PG edges.

Algorithm 1: The algorithm applied by PREC-C (without any context)

Function $precc(pg \in BCPGs) \rightarrow 2^{RdfTriples}$**:**
  $rdf \leftarrow \{\}$
  forall $n \in N_{pg}$ **do**
    $rdf \leftarrow rdf \cup \{(n, rdf{:}type, pgo{:}Node)\}$
    forall $label \in labels_{pg}(n)$ **do**
      $template \leftarrow p0NodeLabelRules(label)$
      $rdf \leftarrow rdf \cup build(template, pg, n)$
  forall $e \in E_{pg}$ **do**
    $rdf \leftarrow rdf \cup \{(e, rdf{:}type, pgo{:}Edge)\}$
    $label \leftarrow$ the only value in the set $labels_{pg}(e)$
    $template \leftarrow p0EdgeRules(label)$
    $rdf \leftarrow rdf \cup build(template, pg, e)$
  forall $p \in P_{pg}$ **do**
    $(key, \_) \leftarrow propdetails_{pg}(p)$
    $template \leftarrow p0PropertyRules(key)$
    $rdf \leftarrow rdf \cup build(template, pg, p)$
  return rdf

[1] An atomic RDF term is any term that is not an RDF Triple: either a literal, an IRI or a blank node. In RDF 1.1, the recursion depth will never exceed 1. In RDF-star, however, β may recurse further.

[2] The C in PREC-C stands for Context. However, in this first version, the user provides no (0) context, so the algorithm falls back to PREC-0 rules.

Algorithm 2: The *build* function for PREC-C

```
   /* Transform the placeholders in the template graph t with values in the PG        */
 1 Function build(tps ⊂ Templates, pg ∈ BCPGs, x ∈ N_pg ∪ E_pg ∪ P_pg) → 2^RdfTriples:
 2     triples ← {}
 3     forall tp ∈ tps do
 4         triples ← triples ∪ {β(tp, pg, x)}
 5     return triples
   /* Transform the placeholders in the triple or term t with values in the PG        */
 6 Function β(tp, pg, x):
 7     if tp ∈ RdfTriples then
 8         (s, p, o) ← tp
 9         return (β(s, pg, x), β(p, pg, x), β(o, pg, x))
10     else if tp ∈ L then return tp
11     else if tp ∈ B then raise Error
12     else if tp =?self then return x
13     else if tp =?source then return src_pg(x)
14     else if tp =?destination then return dest_pg(x)
15     else if tp =?holder then return getHolder(x, pg)
16     else if tp =?value then
17         (key, value) = propdetails_pg(x)
18         return toLiteral(value)
19     else
20         assert(tp ∈ I)
21         return tp
   /* Return the entity the property is on                                            */
22 Function getHolder(p ∈ NEP_pg, pg ∈ BCPGs):
23     assert(p ∈ P_pg)
24     holders = {holder | (holder, p) ∈ properties_pg}
25     assert(|holders| = 1)
26     return the only holder in the set holders
```

Algorithm 3: General purpose PREC-0 functions for Algorithm 1

Function $p0NodeLabelRules(label)$**:**
- `/* Build an IRI for the label */`
- $triples \leftarrow \{\}$
- $iri \leftarrow forgeIRI(:label/, label)$
- $triples \leftarrow triples \cup \{(iri, rdfs{:}label, toLiteral(label))\}$
- `/* Add the triples specific to this node */`
- $triples \leftarrow triples \cup \{(?self, rdf{:}type, iri)\}$
- **return** $triples$

Function $p0EdgeRules(label)$**:**
- `/* Build an IRI for the label */`
- $triples \leftarrow \{\}$
- $iri \leftarrow forgeIRI(:label/, label)$
- $triples \leftarrow triples \cup \{(iri, rdfs{:}label, toLiteral(label))\}$
- `/* Add the triples specific to this edge */`
- $triples \leftarrow triples \cup \{(?self, rdf{:}predicate, iri)\}$
- $triples \leftarrow triples \cup \{(?self, rdf{:}subject, ?source)\}$
- $triples \leftarrow triples \cup \{(?self, rdf{:}object, ?destination)\}$
- **return** $triples$

Function $p0PropertyRules(key)$**:**
- $triples \leftarrow \{\}$
- `/* Build an IRI for the property key */`
- $iri \leftarrow forgeIRI(:key/, key)$
- $triples \leftarrow triples \cup \{(iri, rdf{:}type, prec{:}PropertyKey)\}$
- $triples \leftarrow triples \cup \{(iri, rdfs{:}label, toLiteral(key))\}$
- `/* Add the triples specific to this property */`
- $triples \leftarrow triples \cup \{(?self, rdf{:}type, prec{:}PropertyKeyValue)\}$
- $triples \leftarrow triples \cup \{(?holder, iri, ?self)\}$
- $triples \leftarrow triples \cup \{(?self, rdf{:}value, ?value)\}$
- **return** $triples$

Example 10 [PREC-0 on the running example]
Consider the PG in Figure 3.1 using the GPG formalism introduced in Example 2.
The output of the *precc* algorithm in Algorithm 1 is provided in Listing 4.1

Listing 4.1: Output of PREC-0 on the running example

```
# Type of _:n1 (generated by the precc function)
_:n1 rdf:type pgo:Node .

# Label of Tintin (p0NodeLabelRules)
:label/Person rdfs:label "Person" .
_:n1 rdf:type :label/Person .

# Type of _:n2 (generated by the precc function)
_:n2 rdf:type pgo:Node .

# Type of _:e1 (generated by the precc function)
_:e1 rdf:type pgo:Edge .

# TravelsWith edge (p0EdgeRules)
:label/TravelsWith rdfs:label "TravelsWith" .
_:e1 rdf:predicate :label/TravelsWith .
_:e1 rdf:subject _:n1 .
_:e1 rdf:object _:n2 .

# name property of Tintin (p0PropertyRules)
:key/name rdf:type prec:PropertyKey . # Also added by _:p3
:key/name rdfs:label "name" . # Also added by _:p3
_:p1 rdf:type prec:PropertyKeyValue .
_:n1 :key/name _:p1 .
_:p1 rdf:value "Tintin" .

# job property of Tintin (p0PropertyRules)
:key/job rdf:type prec:PropertyKey .
:key/job rdfs:label "job" .
_:p2 rdf:type prec:PropertyKeyValue .
_:n1 :key/name _:p2 .
_:p2 rdf:value "Reporter" .

# name of Snowy (p0PropertyRules)
_:p3 rdf:type prec:PropertyKeyValue .
_:n2 :key/name _:p3 .
_:p3 rdf:value "Snowy" .

# since property of the edge (p0PropertyRules)
:key/since rdf:type prec:PropertyKey .
:key/since rdfs:label "since" .
_:p4 rdf:type prec:PropertyKeyValue .
_:e1 :key/since _:p4 .
_:p4 rdf:value 1978 .
```

4.1.3 Second iteration: context basic support

In this section, the support of the context is added to allow users to customize the transformation. Indeed, the transformation presented in the previous Section 4.1.2 can not be customized, despite the motivations of building PREC-C being to be able to do it.

PREC-C Context

A *PREC-C context* is a function that maps *selectors* to template graphs. The set of PREC-C contexts is *Ctxc*.

Definition 15 [PREC-C Placeholders]
The set of PREC-C placeholders is as follows:

- The node placeholders, *i.e.* the members of the set PN, are $?self$, $?source$, $?destination$, $?holder$. They are respectively placeholders for the NEP itself, the source node of an edge, the destination node of an edge and the holder of a property.
- The IRI placeholders, *i.e.* the members of the set PI, are $?nodeLabelIRI$, $?edgeIRI$ and $?propertyIRI$. They are respectively placeholders for an IRI that represent the label of a node, the label of an edge and a property key.
- The literal placeholders, *i.e.* the members of the set PL, are $?label$, $?key$ and $?value$. They are respectively placeholders for the string representation of a node or edge label, the string representation of a property key, and the property value.

Definition 16 [Selector of NEPs]
A *selector* lets the user select NEPs depending on their type and their position in the PG relatively to NEPs they are linked to.

There are three kinds of selectors, one per kind of NEP:

- *Node selectors* are pairs ("*node*", *theLabel*) where *theLabel* is a node label.
- *Edge selectors* are tuples ("*edge*", *label*, *srcLabels*, *destLabels*) where *label* is the edge label, *srcLabels* is the list of all source node labels and *destLabels* is the list of all destination node labels.
- *Property selectors* are tuples ("*property*", *key*, *holderKind*, *holderLabels*) where *key* is the property key, *holderKind* is the kind of the property holder and *holderLabels* is the list of labels of the holder or the singleton composed of the holder key if it is a property key.

The set of all selectors is denoted *Sel* and is formally defined as:

$$Sel = \bigcup \begin{cases} \{\text{“}node\text{”}\} \times Str & \text{Node selectors} \\ \{\text{“}edge\text{”}\} \times Str \times 2^{Str} \times 2^{Str} & \text{Edge selectors} \\ \{\text{“}property\text{”}\} \times Str \times \{\text{“}node\text{”}, \text{“}edge\text{”}, \text{“}property\text{”}\} \times 2^{Str} & \text{Property selectors} \end{cases}$$

Definition 17 [Computing the selector of a NEP]
Let *pg* be a PG.

A node can be selected by multiple node selectors. The list of all selectors of a node n is $\{(\text{“}node\text{”}, theLabel) \mid theLabel \in labels_{pg}(n)\}$. Note that nodes that have no label do not have any selector.

Edges and properties have one and only one selector, that can be computed by the $select_{pg} : E_{pg} \cup P_{pg} \to Sel$ function.

For edges, the $select_{pg}$ function computes the labels and the edge, its source and its

destination:

$$\forall e \in E_{pg}, select_{pg}(e) = (\text{“}edge\text{”}, label, srcLabels, destLabels) \text{ with:}$$
$$\{label\} = labels_{pg}(e),$$
$$srcLabels = labels_{pg}(src_{pg}(e)),$$
$$destLabels = labels_{pg}(dest_{pg}(e))$$

For properties, the $select_{pg}$ function computes the key of the property, the kind of the holder and its labels/property key:

$$\forall p \in P_{pg}, select_{pg}(p) = (\text{“}property\text{”}, key, holderKind, holderLabels) \text{ with:}$$
$$(key, \_) = propdetails_{pg}(p),$$
$$holderKind = kind_{pg}(getHolder(p, pg)),$$
$$holderLabels = \begin{cases} labels_{pg}(getHolder(p, pg)) & \text{if } getHolder(p, pg) \in N_{pg} \cup E_{pg} \\ \{propdetails_{pg}(getHolder(p, pg))\} & \text{if } getHolder(p, pg) \in P_{pg} \end{cases}$$

Example 11 [Selectors]
In the PG that serves as a running example in Figure 3.1:

- The selector of the “Person” label of Tintin is (“*node*”, “*Person*”).
- The selector of the “name” property of Tintin is (“*property*”, “*name*”, “*node*”, {“*Person*”}).
- The selector of the “name” property of Snowy is (“*property*”, “*name*”, “*node*”, {}).
- The selector of the “job” property of Tintin is (“*property*”, “*job*”, “*node*”, {“*Person*”})
- The selector of the only edge in the PG is (“*edge*”, “*TravelsWith*”, {“*Person*”}, {}).
- The selector of the “since” property of the edge is (“*property*”, “*since*”, “*edge*”, {“*TravelsWith*”})

The PG in Figure 3.2 only has one node on which there are two “name” properties: one without any meta-property and one with a “type” meta-property. The selector of the “type” meta-property is (“*property*”, “*type*”, “*property*”, {“*name*”}).

Definition 18 [PREC-C contexts]
A *PREC-C context* is a total function that maps all selectors to template graphs:

$$ctx \in Ctxc \Leftrightarrow ctx : Sel \to 2^{Templates}$$

The template graphs must be *valid i.e.* placeholders reserved to a specific kind should only be used in template graphs used by this kind. The list of allowed placeholders is:

- In template graphs used by node label selectors: $?self$.
- In template graphs used by edge selectors: $?self$, $?source$ and $?destination$.
- In template graphs used by property selectors: $?self$, $?holder$ and $?value$.
- The placeholders $?nodeLabelIRI$, $?edgeIRI$, $?propertyIRI$, $?key$ and $?label$ are not allowed in any template graph: they are used for internal purposes discussed later in Section 4.2.2.

Algorithm 4: The algorithm applied by PREC-C with a context

```
1  Function precc(pg ∈ BCPGs, ctx ∈ Ctxc) → 2^RdfTriples:
2      rdf ← {}
3      forall n ∈ N_pg do
4          rdf ← rdf ∪ {(n, rdf:type, pgo:Node)}
5          forall label ∈ labels_pg(n) do
6              selector ← ("node", "label")
7              template ← ctx(selector)
8              rdf ← rdf ∪ build(template, pg, n)
9      forall e ∈ E_pg do
10         rdf ← rdf ∪ {(e, rdf:type, pgo:Edge)}
11         label ← the only value in the set labels_pg(e)
12         selector ← select_pg(e)
13         template ← ctx(selector)
14         rdf ← rdf ∪ build(template, pg, e)
15     forall p ∈ P_pg do
16         (key, _) ← propdetails_pg(p)
17         selector ← select_pg(p)
18         template ← ctx(selector)
19         rdf ← rdf ∪ build(template, pg, p)
20     return rdf
```

Definition 19 [The PREC-0 context]
The PREC-0 context ctx_{p0} is defined as:

$$ctx_{p0}(selector) = \begin{cases} p0NodeLabelRules(label) & \text{if } selector = (\text{“}node\text{”}, label) \\ p0EdgeRules(label) & \text{if } selector = (\text{“}edge\text{”}, label, \_1, \_2) \\ p0PropertyRules(key) & \text{if } selector = (\text{“}property\text{”}, key, \_1, \_2) \end{cases}$$

ctx_{p0} is the context implicitly used in Algorithm 1, *i.e.* the *precc* function in Algorithm 1 is the same function as the *precc* function in Algorithm 4 when the value of the second parameter is ctx_{p0}.

Discussion about selectors

Selectors do not only hold the kind and the label of the NEPs they select, but also some information about the NEPs the selected NEP is connected to.

This enables to use different template graphs on NEPs with the same kind and label. For example, consider the running example PG. The name of a person and the name of an animal may use different IRIs to provide a more precise semantics.

Property selectors could be further extended by including more information about the holder, the holder of the holder, ... like in Definition 20 instead of using only information about the holder. Doing so would not really change the formal definitions, as only the *Sel* set and the *select* function would need to be redefined. The selector definition presented in Definition 16 is the one that has been implemented.

Definition 20 [Possible property selector extension]
For a given PG pg and a given property p, the $select_{pg}(p)$ function could be redefined as $select_{pg}(p) = (\text{“}property\text{”}, subselect_{pg}(p))$ with

$$subselect_{pg}(m) = \begin{cases} labels_{pg}(m) & \text{if } m \in N_{pg} \\ \begin{pmatrix} subselect_{pg}(src_{pg}(m), \\ subselect_{pg}(dest_{pg}(m)) \end{pmatrix} & \text{if } m \in E_{pg} \\ \begin{pmatrix} key, \\ subselect_{pg}(getHolder_{pg}(m)) \end{pmatrix} & \text{if } m \in P_{pg} \\ & \text{with } (key, \_1) = propdetails_{pg}(m) \end{cases}$$

Note that this definition is not actually used. The rest of the chapter sticks to the selection definition provided in Definition 16.

4.1.4 Third iteration: supporting *?self*-less templates

Motivation

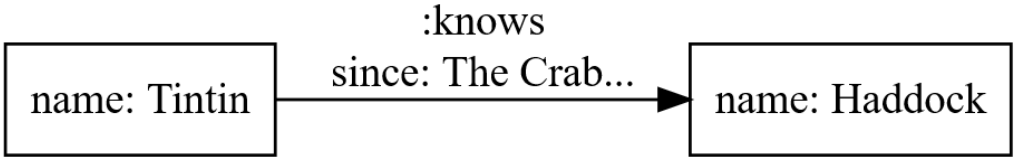


Figure 4.1: A simplified PG where Tintin knows Haddock

Listing 4.2: The expected RDF graph from the PG in Figure 4.1

```
_:tintin foaf:name "Tintin" .
_:haddock foaf:name "Haddock" .
_:tintin foaf:knows _:haddock .
<< _:tintin foaf:knows _:haddock >> ex:since "The Crab ..." .
```

Consider the PG in Figure 4.1. A fairly idiomatic way to represent the same information in RDF is exposed in Listing 4.2. To produce the later RDF graph using PREC-C, it is expected that the following context can be used:

$$ctx(sel) = \begin{cases} \{(?holder, foaf{:}name, ?value)\} & \text{if } sel = (\text{“}property\text{”}, \text{“}name\text{”}, \_1, \_2) \\ \{(?holder, ex{:}since, ?value)\} & \text{if } sel = (\text{“}property\text{”}, \text{“}since\text{”}, \_1, \_2) \\ \{(?source, foaf{:}knows, ?destination)\} & \text{if } sel = (\text{“}edge\text{”}, \text{“}knows\text{”}, \_1, \_2) \\ \emptyset & \text{otherwise} \end{cases}$$

However, by following the exposed algorithms, the produced RDF graph is not the one exposed in Listing 4.2 but the one exposed in Listing 4.3. Note that the fourth triple is not the same: instead of using the new identity of the edge, `_:tintin foaf:knows _:haddock`, the PREC-C engine continues to use the edge blank node.

Listing 4.3: The expected RDF graph currently produced by PREC-C from the PG in Figure 4.1 and the given context

```
_:tintin foaf:name "Tintin" .
_:haddock foaf:name "Haddock" .
_:tintin foaf:knows _:haddock .
_:p3 ex:since "The Crab ..." .
```

When a template is written, the $?self$ placeholder may not be in the template graph. If $?self$ is not in the template graph of an edge or a property, the term that plays its role, which may be a triple considering RDF-star, must be used as the $?holder$ value instead of the NEP identity when converting the properties it holds.

Specifying the new self identity

For this purpose, we now require contexts to not only provide the template graph, but also the new self identity.

Definition 21 [PREC-C Context with explicit self]
A PREC-C context $ctx \in Ctxc$ is now a function that maps all selectors to a template graph and the *new self identity*: $ctx : Sel \rightarrow (2^{Templates}, I \cup PN \cup Templates)$

The template graph still has to be *valid* as described by Definition 18. Moreover, the *new self identity* associated to node selectors is restricted to always be $?self$[a].

The ctx_{p0} function is now defined as:

$$ctx_{p0}(selector) = \begin{cases} (p0NodeLabelRules(label), ?\mathbf{self}) & \text{if } selector = (\text{“}node\text{”}, label) \\ (p0EdgeRules(label), ?\mathbf{self}) & \text{if } selector = (\text{“}edge\text{”}, label, \_1, \_2) \\ (p0PropertyRules(key), ?\mathbf{self}) & \text{if } selector = (\text{“}property\text{”}, key, \_1, \_2) \end{cases}$$

[a] A node can be selected by multiple labels. If the context assigned a different “new self identity” to different selectors of the same node, there would be an ambiguity on the identity of that node in the RDF graph. Therefore, the “self identity” of a node may not be changed.

To account for the fact that the context now returns two values, Algorithm 4 is extended into Algorithm 5 with the following modifications. First, all lines $template \leftarrow ctx(selector)$ are replaced by $(template, \_) \leftarrow ctx(selector)$. Then, the property loop is updated to use a new function *resolveHolder*. This functions looks for a given NEP its new self identity template, and instantiates it with the build function to replace the placeholders with the actual values, for example to replace $?self$ with the NEP, or $?source$ with the source node. The recursion handles the possible meta-properties. The result of the *resolveHolder* function is used to replace the value of $?holder$ in the template graph with the actual holder identity. The *substitute* function is introduced in Algorithm 6: it takes as input a template triple and a pair composed of a source term and a destination term, and outputs the same template graph where all occurrences of the source term have been replaced with the destination term.

Explicitly defining the self identity might be a tedious task, and may be error-prone in practice. A utility function that computes the most likely *new self identity* is provided.

Definition 22 [Deducing the new self identity]
The $deduceSelf : Templates \rightarrow I \cup PN \cup Templates$ function tries to find the most likely new self identity and is defined as:

Algorithm 5: The algorithm applied by PREC-C with a context

```
1  Function precc(pg ∈ BCPGs, ctx ∈ Ctxc) → 2^RdfTriples:
2      rdf ← {}
       /* Same node and edge loops as in Algorithm 4 with the previously discussed
          modification                                                          */
3      forall p ∈ P_pg do
4          (key, _) ← propdetails_pg(p)
5          selector ← select_pg(p)
6          (template, _) ← ctx(selector)
7          myHolder = resolveHolder(holder_pg(p), pg, ctx)
8          template = substitute(template, (?holder, myHolder))
9          rdf ← rdf ∪ build(template, pg, p)
10     return rdf
11 Function resolveHolder(nep, pg, ctx):
12     if nep ∈ N_pg then
           /* Nodes can not be remapped                                        */
13         return nep
14     selector ← select_pg(p)
15     (_, selfIdentity) ← ctx(selector)
16     if nep ∈ E_pg then
           /* Instantiate the variables in the identity and return it          */
17         return β(selfIdentity, pg, nep)
18     else
19         assert(nep ∈ P_pg)
           /* We need to resolve the holder of the holder                      */
20         myHolder ← holder_pg(nep)
21         holderSelf ← resolveHolder(myHolder, pg, ctx)
           /* Instantiate the new self identity and return it                  */
22         selfIdentity ← substitute(selfIdentity, (?holder, holderSelf))
23         return β(selfIdentity, pg, nep)
```

Algorithm 6: The substitute function

```
1  Function substitute(template, (from, to)):
2      if template = from then
3          return to
4      else if template ⊆ Templates ∪ RdfTriples then
5          return {substitute(triple, (from, to)) | triple ∈ template}
6      else if template ∈ Templates ∪ RdfTriples then
7          (s, p, o) ← template
8          subS ← substitute(s, (from, to))
9          subP ← substitute(p, (from, to))
10         subO ← substitute(o, (from, to))
11         return (subS, subP, subO)
12     else
13         return template
```

$$deduceSelf(template) = \begin{cases} s = ?self & \text{if } \exists t \in template, s \in t \\ s = (?source, iri, ?destination) & \text{else if } \exists iri \in I, t \in template, s \in t \\ s = (?destination, iri, ?source) & \text{else if } \exists iri \in I, t \in template, s \in t \\ s = (?holder, iri, ?value) & \text{else if } \exists iri \in I, t \in template, s \in t \\ \text{undefined} & \text{otherwise} \end{cases}$$

The *deduceSelf* function has to guess the intent of the template designer, and more precisely what may identify an edge or a property. It may be identified by:

- A unique identifier. The only term that may generate a unique identifier in a template graph is *?self* as it generates a blank node that is specific for each NEP. No other placeholder can generate a blank node or IRI specific to this NEP.
- Not relying on a unique identifier: A unique identifier is not required because:
  - The edge is considered to be **edge-unique**, *i.e.* an edge such that there is at most one edge with this given label between two nodes. In this case, it is very likely that a direct triple will be built, with either the source or the destination as the subject, the other one as the object and an IRI that describes the label as the predicate.
  - The property follows the set insertion strategy of Gremlin, *i.e.* all key-value pairs are unique for a given holder and for this given key. In this case, it is very likely that the template graph contains a triple with the holder of the property as the subject, a predicate that identifies the property key, and the property value as the object.

For deterministic purpose and to keep the formalism as simple as possible, if multiple triples match the same requirements of the *deduceSelf* function, the triple (1) that is the less nested in the template triples, (2) with the smaller *iri* member in lexicographical order is chosen by *deduceSelf*.

The current implementation actually behaves differently: all candidates are picked, and every time the value of self is required for a holder, the template graph is instantiated for each found *new self identity*. However, having multiple candidates with the same priority should probably be considered bad practice. It seems preferable to have a template graph as small as possible with as few as possible redundancy, and infer later the redundancy with an inference language like RDFS or OWL.

4.1.5 The final version of the PREC-C algorithm

Algorithm 7: The main PREC-C algorithm

Function $precc(pg \in BCPGs, ctx \in Ctxc) \rightarrow 2^{RdfTriples}$**:**
- $rdf \leftarrow \{\}$
- **forall** $n \in N_{pg}$ **do**
  - $rdf \leftarrow rdf \cup \{(n, rdf{:}type, pgo{:}Node)\}$
  - **forall** $label \in labels_{pg}(n)$ **do**
    - $selector \leftarrow (\text{“}node\text{”}, label)$
    - $(template, \_) \leftarrow ctx(selector)$
    - $rdf \leftarrow rdf \cup build(template, pg, n)$
- **forall** $e \in E_{pg}$ **do**
  - $rdf \leftarrow rdf \cup \{(e, rdf{:}type, pgo{:}Edge)\}$
  - $label \leftarrow$ the only value in the set $labels_{pg}(e)$
  - $selector \leftarrow select_{pg}(e)$
  - $(template, \_) \leftarrow ctx(selector)$
  - $rdf \leftarrow rdf \cup build(template, pg, e)$
- **forall** $p \in P_{pg}$ **do**
  - $(key, \_) \leftarrow propdetails_{pg}(p)$
  - $selector \leftarrow select_{pg}(p)$
  - $(template, \_) \leftarrow ctx(selector)$
  - $myHolder = resolveHolder(holder_{pg}(p), pg, ctx)$
  - $template = substitute(template, (?holder, myHolder))$
  - $rdf \leftarrow rdf \cup build(template, pg, p)$
- **return** rdf

Algorithm 7 exposes the final version of the PREC-C function. This function expects two parameters: the PG to convert and the PREC-C context.

The PREC-C context *ctx* is a function such as described by Definition 21

Selectors are as defined in Definition 16 and the associated function *select* is defined in Definition 20. The *build* / β functions are unchanged from Algorithm 2[3].

4.1.6 Going further

The PREC-0 ontology

When the PREC-C algorithm is used in the first version presented in Section 4.1.2, an RDF graph that complies with the schema presented in Figure 4.2 is produced.

[3]Appendix A proposes a rewriting of the β function.

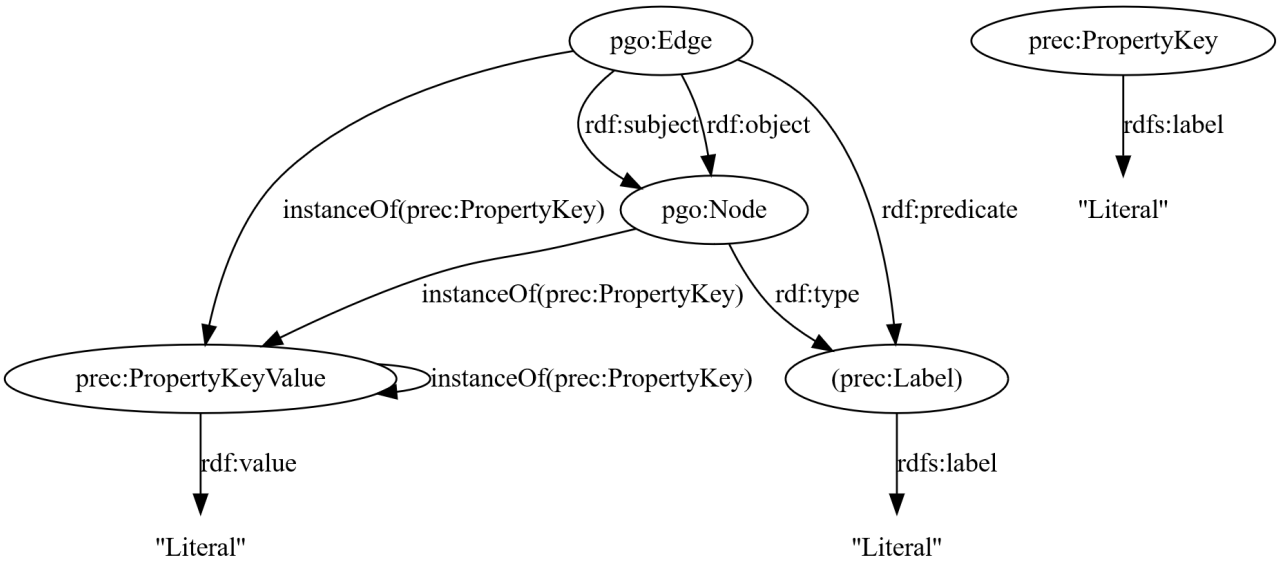


Figure 4.2: The schema of the PREC-0 graphs

Nodes and edges of the PG are all represented in the RDF graph by blank nodes, which are instances of the types `pgo:Node` and `pgo:Edge`, respectively. Nodes are linked to their labels through the `rdf:type` label. Edges are represented with the classic RDF reification: they are connected to their starting node through `rdf:subject`, their destination node through `rdf:object` and their label by using `rdf:predicate`. Each label (as a resource) is linked to its value as a literal using the `rdfs:label` predicate. Note that the `prec:Label` type is not given to these resources, and is only shown in the Figure in parentheses to help to understand. Nodes and edges are linked to the property they hold through IRIs forged using the property key. These forged IRIs are typed with the `prec:PropertyKey` type, linked to their label as a literal with the `rdfs:label` predicate, and represented in the figure by the "instanceOf(prec:PropertyKey)" labels on the edges. Meta-properties are connected to properties using the same mechanism. The raw value of the property as a literal is linked to the property blank node through the `rdf:value` predicate.

The PREC-0 schema has been built with two major and one minor design goals:

- It should reuse as much as possible RDF constructions. This is achieved by using the standard RDF reification for edges, and using terms in the `rdf` and `rdfs` namespaces. The only terms not in these namespaces are `pgo:Node` and `pgo:Edge` which we borrowed from the Property Graph Ontology, the `prec:PropertyKey` and `prec:PropertyKeyValue` types and the IRIs that are forged for labels and property keys.
- It should be reversible, *i.e.* it allows reconstructing the original PG from the RDF graph. It achieves this by being a literal description of the converted Property Graph. `rdf:subject` means "starting PG node", `rdf:object` means "destination PG node", `rdf:type` and `rdf:predicate` respectively mean node label and edge label. To avoid some NEP collapsing, *i.e.* two NEPs having the same representation in the produced RDF graph making them indistinguishable[4], no blank node from the original BPG is discarded during the conversion.

[4]For example two properties with the same key being converted to the same triples

For a given forged IRI, its "contextual triples" are always the same, *e.g.* a property key IRI forged for two different properties has the same property key for both properties.

- The minor design goal is that the PREC-0 ontology should be as close as possible to an idiomatic RDF graph, under the constraint of relying on a one-size-fit-all transformation. This is partly answered by the first major design goal, *i.e.* using as much as possible usual RDF constructs. This is also answered by the fact that the forged property keys are used in predicate position. Indeed, they are very likely to be mapped to existing predicates like `foaf:name`. On the opposite, consider a construction like PGO as exposed in Figure 4.3 where generic predicates are used and the name of the property is in the object position of a triple whose subject is the property. With such a construction, the generated RDF graph structure is more different from the final expected RDF graph than the RDF graph produced by the PREC-0 ontology. Moreover, while PGO tends to directly use the labels and property keys in object position as a literal, by using forged label IRIs and property keys, PREC-0 lets the user map these forged terms to real world terms, for example by later using `owl:equivalentProperty` on the produced RDF graph.

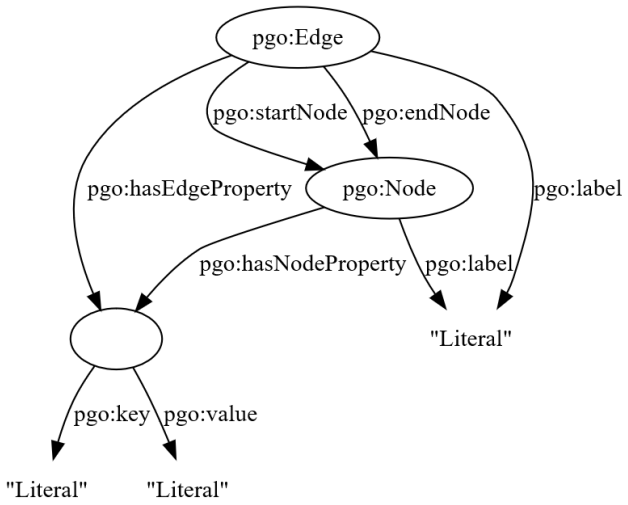


Figure 4.3: The schema of the Property Graph Ontology

Reversibility of PREC-C contexts

The reversibility of a PREC-C context *ctx* is defined as the possibility to compute back any PG *pg* from the value of $precc(pg, ctx)$ and the PREC-C context *ctx*.

In this section, reversibility will only be discussed informally, and in particular what may be good criteria to decide quickly if a context may be reversible or not. A formal definition of reversibility will be provided later in Section 5.5.1.

Trivially, not all contexts are reversible. Consider a context that produces the empty template graph for all selectors. When using the *precc* function, the output will always be the empty RDF graph. From the empty RDF graph, it is impossible to compute back the PG as multiple PGs (in fact, any PG) may have produced it.

Example 12 [Different template graphs to model that Tintin travels with Snowy] Consider a PG in which there is an edge with the label "travelsWith" between Tintin and

Snowy, with no property. Examples of template graphs that could be used to model this edge in RDF are:

1. $\{(?self, ex{:}traveler, ?source), (?self, ex{:}traveler, ?destination)\}$ with $?self$ as the "self identity".
   - This template graph is not reversible because it is impossible to find which node was the source and which node was the destination as the $ex{:}traveler$ predicate is used for both.
2. $\{(?self, ex{:}traveler1, ?source), (?self, ex{:}traveler2, ?destination)\}$ with $?self$ as the "self identity".
   - This template graph is reversible: if a triple $(\_{:}travelers, ex{:}traveler1, \_{:}tintin)$ is mapped to the template triple $(?self, ex{:}traveler1, ?source)$, then $\_{:}travelers$ is an edge of the PG and $\_{:}tintin$ is a node of the PG. A similar process can be applied for the destination. All properties will be attached to $?self$ so properties will properly be connected to the edge in the output RDF graph.
3. $\{(?source, ex{:}travelsWith, ?destination)\}$ with $?self$ as the "self identity".
   - This template graph is not reversible: the properties of the edge will not be related with the source and the destination of the edge in the output RDF graph. This case is the one that led us to add the notion of "self identity" in Section 4.1.4.
4. $\{(?source, ex{:}travelsWith, ?destination)\}$ with the whole template triple as the "self identity".
   - This template graph looks reversible: if a triple $(\_{:}tintin, ex{:}travelsWith, snowy)$ is mapped to the template triple $\{(?source, ex{:}travelsWith, ?destination)\}$, then $\_{:}tintin$ and $snowy$ are nodes, and there is another edge in the PG, for which the source is $\_{:}tintin$ and the destination is $snowy$. While the exact original PG cannot be computed back, we can build an isomorphic PG.
   - However, this assumes that between any two PG nodes, there is at most one PG edge with the label "travelsWith". If that is not the case, then multiple PG edges will be collapsed into a single RDF triple, and the information about how many such edges were in the PG, and the distribution of their properties, will be lost.
5. The empty template graph with any "self identity".
   - This template graph is trivially not reversible because the source and the destination are not related with the edge in the RDF graph, especially in the case where there are no properties for the edge.
   - Using an empty template graph with $\{(?source, ex{:}travelsWith, ?destination)\}$ as the "self identity" may find some uses in situations where we are guaranteed that there will be at least one property, but we do not want to assert the triple, and do not want to use other constructs. In the example of a "TravelsWith" edge, it may be because all edges have a "until" property that limits the truthiness of the triple to a time period.

From Example 12, we can see that there are (at least) two important criteria to determine if a template graph used in a PREC-C context is reversible:

- The "self identity" of each template triple must be unique.
- The "self identity" must be related in the RDF graph with all information that are relevant, in a non-ambiguous manner:
  - For edges, the source and the destination must be in the template graph.
  - For properties, the holder and the property value must be in the template graph.
  - In both case, the "self identity" and the triples of the template graph must be consistent (unlike the fourth template graph of Example 12)

Another question that is not considered in this section is how to determine the selectors that selected the NEPs, and how to find in a non-ambiguous manner the information, *i.e.* how to map the triples in the produced RDF graph to the template triple that produced it. While this question is left for future works in the case of PREC-C, in the following Chapter 5, that presents another converter named PRSC, this question will be formally answered.

4.1.7 Complexity analysis

We now discuss the complexity of the *precc* algorithm as presented in Algorithm 7.

Functions considered constant

For a given PG pg, the complexity of all functions that define the PG, *i.e.* src_{pg}, $dest_{pg}$, $labels_{pg}$, $getHolder_{pg}$ and $propdetails_{pg}$ are considered constant.

The complexity of the function *toLiteral* is also considered constant.

Evaluating if something is a member of a given set, for example if an entity is part of the set N_{pg}, is generally considered to be constant time thanks to hash maps[5].

Considered metrics

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

- The number of NEPs in the PG, denoted $NbOfNEPs$.

$$NbOfNEPs = |N_{pg} \cup E_{pg} \cup P_{pg}|$$

- The maximal number of labels in the PG, denoted $TypeComplexity$:

$$TypeComplexity = \max_{n \in N_{pg}}(|labels_{pg}(n)|)$$

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

$$BiggestTemplateSize = \max_{selector \in Dom(ctx)}(|tps \mid (tps, \_1) = ctx(selector)|)$$

- The depth of the most nested meta property, denoted $MetaDepth$.

$$MetaDepth = \max_{p \in P}(\{n \in \mathcal{N} \mid \exists m = getHolder^n(p)\})$$

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

- The cost of calling the *ctx* function *CtxCost*. As *ctx* is a total function, it is difficult to evaluate the precise complexity without its actual definition. The function could serve very generic templates like the ctx_{p0} function, or very specific templates for multiple selectors.

In RDF-star, quoted triples can be used as subject or object of other triples, without limit on how deeply triples can be nested. In practice, however, it is rare to have more than one level of nesting. Usually, users are expected to use atomic RDF triples like `_:tintin :travelsWith _:haddock` or to use RDF-star triples with a depth of one like `<< _:tintin :travelsWith _:haddock >> :since 1978`. We therefore consider the *depth* of any triple to be bound by a constant. As a consequence, in all functions processing terms and triples recursively (such as β in Algorithm 2 or *substitute* in Algorithm 6), we can ignore the recursion depth in the complexity analysis.

In all complexity analyses, all metrics but *MetaDepth* are considered non null. Indeed, if there are no NEPs or if the biggest template graph is empty, the produced RDF graph will be empty so this case is not interesting. In the case of *MetaDepth*, we consider that by convention if there is no meta property, its value is equal to zero.

Complexity of the utility functions

We now discuss the complexity of the utility functions from which the *precc* function is defined.

substitute The *substitute* function can be called on graphs, on triples and on terms. Its complexity is constant for terms, so as discussed in Section 4.1.7, its complexity is also constant on triples. As the substitute function calls itself on all triples when called with a list of triples, and as it is always called on a graph to transform, produced from a template graph, its complexity is linear with the size of the biggest template graph: $\mathcal{O}(BiggestTemplateSize)$.

select The *select* function can be called in constant time: it calls all functions that characterize a PG at most twice, and do not call any other function:

- For edges, only the *labels*, *src* and *dest* functions are called, and they are called once.
- For properties, the *propdetails* and the *getHolder* functions are called once for the property itself. At most two set membership operations are performed to find the holder kind, which are performed in constant time. Depending on the kind of the holder, either the *labels* function is called once, or the *propdetails* function is called an extra time.

resolveHolder The *resolveHolder* is called recursively until it finds a node or an edge. Calls on a node are trivially done in a constant time. Calls on edges and properties first call the *select* function which is called in constant time. Then for properties, it calls the *getHolder* function, the *resolveHolder* and the *substitute* function on a template term; the first is performed in constant time as mentioned in Section 4.1.7 and the third is also performed in constant time as mentioned previously.

As all calls done in the *resolveHolder* function are done in constant time, and as the recursion depth is equal to *MetaDepth*, the complexity of the *resolveHolder* is linear with the depth of the most nested property.

Complexity of the *build* / β functions The β function consists in calling the function recursively on template triples until an atomic RDF term is found. Then, several checks are performed on the term, and either the term itself is returned or a function with a constant cost is called. As the recursion depth of recursive functions on terms is considered negligible, the complexity of the β function is also considered constant.

The *build* function consists in calling the β function on each template triple. Its complexity is $\mathcal{O}(BiggestTemplateSize)$ as the β has a constant complexity.

Complexity of the *precc* function

The *precc* function consists in three loops: one per kind of NEP.

- For nodes, there is a loop on each label (the number of loops is equal to the type complexity). For each label, there are one call to the *ctx* function (linear with its given metric) and one call to the *build* function (linear with the size of the biggest template graph). The cost of the iteration of each node is $TypeComplexity * (CtxCost + BiggestTemplateSize)$.
- For edges, there is one call to the $select_{pg}$ (constant complexity), *ctx* (linear with its given metric) and the *build* function (linear with the size of the biggest template graph). The cost of the iteration of each edge is $CtxCost + BiggestTemplateSize$.
- For properties, there is one call to the *propdetails* function (constant complexity), to the $select_{pg}$ (constant complexity), to the *resolveHolder* (linear with the depth of the most nested property) and the *getHolder* (constant complexity), to the *ctx* function (linear with its own metric), to the *substitute* function (linear with the size of the biggest template graph), and to the *build* function (linear with the size of the biggest template graph). The cost of the iteration on each property is $MetaDepth + CtxCost + BiggestTemplateSize$.

The final complexity of the *precc* function is:

$$
\begin{aligned}
&\mathcal{O}(NbOfNEPs * (TypeComplexity * CtxCost * BiggestTemplateSize) \\
&\quad + NbOfNEPs * (CtxCost + BiggestTemplateSize) \\
&\quad + NbOfNEPs * (MetaDepth + CtxCost + BiggestTemplateSize)) \\
=&\mathcal{O}(NbOfNEPs \\
&\quad * (TypeComplexity * (CtxCost + BiggestTemplateSize) + MetaDepth))
\end{aligned}
$$

We notice that the complexity of *precc* is linear in each of the metrics we introduced. Implementations can therefore be expected to scale well.

4.2 Implementation of PREC-C

Section 4.1 provides a formal definition of the PREC-C transformation. This section connects the formal definition with the actual API and implementation of PREC-C, and in particular how PREC-C contexts are expressed.

PREC-C contexts are described by the user using the Turtle language, the PREC ontology and the various placeholders that are represented by IRIs in the reserved `pvar` namespace. Table 4.1 gives an overview of the different rule types, corresponding to each kind of element, and their related predicates.

Table 4.1: Rule types and the predicates related to them in the PREC-C ruleset

Type	`prec:NodeLabelRule`	`prec:EdgeRule`	`prec:PropertyRule`
Selector			
Target	Nodes selected by (“*node*”, *label*)	Edges selected by (“*edge*”, *label*, *srcLabels*, *destLabels*)	Properties selected by (“*property*”, *key*, *holderKind*, *holderLabels*)
Required predicates	`prec:label`: The value of *label*	`prec:label`: The value of *label*	`prec:propertyKey`: The value of *key*
Optional predicates	*None*	`prec:sourceLabel`: The subset values of *srcLabels* `prec:destinationLabel`: The subset values of *destLabels*	`prec:label`: The subset values of *holderLabels* `prec:onKind`: The value of *holderKind*
Priority	`prec:priority`: If several rules match a selector, the rule with the lowest priority will be used		
Production			
Specify the used extended template graph	`prec:templatedBy`: The name of the extended template graph to use		
Template graph	`prec:produces`: The template triples in the template graph as embedded triples		

Expected placeholders in the template graph	`pvar:nodeLabelIRI`
	`pvar:self`
	`pvar:label`
		`pvar:edgeIRI`
		`pvar:self`
		`pvar:source`
		`pvar:destination`
		`pvar:label`
			`pvar:propertyIRI`
			`pvar:property`
			`pvar:holder`
			`pvar:value`
			`pvar:key`

New *?self* ***identity***	*N/A*	`prec:edgeIs`	`prec:entityIs`
Predicate for IRI description	`prec:nodeLabelIRI`	`prec:edgeIRI`	`prec:propertyIRI`

4.2.1 The PREC-C ontology

Table 4.1 exposes the three different types of rules that exist in the PREC-C ruleset: `NodeLabelRule`s, `EdgeRule`s and `PropertyRule`s. Each type of rule corresponds to a specific kind of selector and applies to all the NEPs that match their conditions. The table lists for each rule type the list of related predicates. Note that the listed placeholders for each type should be used in the template graph, and can be used in any position in the template triples.

Selector predicates

The upper part of Table 4.1 describes the selectors that are concerned by the rule. It is worth noting that, although `prec:NodeLabelRule`s always match exactly one node selector, `prec:EdgeRule`s and `prec:PropertyRule`s may match several selectors:

- A `prec:EdgeRule` matches any edge selector whose *srcLabels* contains all values of `prec:sourceLabel` (and possibly others), and whose *destLabel* contains all values of `prec:destinatonLabel`.
- A `prec:PropertyRule` matches any property selector whose *holderKind* is one of the values of `prec:onKind`, and whose *holderLabels* contains all the values of `prec:label`. If no value is provided for `prec:onKind`, the rule is considered to match all possible values. It is worth noting that for meta-properties, the value of `prec:onKind` is `"property"` and *holderLabels* must have at most one value which is the parent property key. As a consequence, a rule for meta-properties with several values for `prec:label` will fail to match any property because a property can only have one key.

However, a selector may be concerned by multiple rules. Consider the PREC-C context in Listing 4.4. Two edge rules select edges with the "*travelsWith*" label: the `:TravelWithRulePerson` rule and the `:TravelWithRuleAnimal`. The only thing that distinguishes them is that one can only be applied to edges with a source node with the "Person" label and the other one to edges with a destination node with the "Animal" label. However, what happens if a "travelsWith" edge starts on a person and ends on an animal? In other words, which rule should be used by the engine for the selector ("*edge*", "*TravelsWith*", {"*Person*"}, {"*Animal*"}), and by extension, any selector for which the third and fourth members are a super-set of this selector?

Listing 4.4: Two rules for edge selectors on the "since" edge label

```
:TravelWithRulePerson a prec:EdgeRule ;
    prec:label "travelsWith" ;
    prec:sourceLabel "Person" ;
    prec:templatedBy prec:RDFReification .

:TravelWithRuleAnimal a prec:EdgeRule ;
    prec:label "travelsWith" ;
    prec:destinationLabel "Animal" ;
    prec:templatedBy prec:DirectTriples .
```

The `prec:priority` predicate enables to pick the rule that should be applied in case multiple rules are applicable. The rule with the highest value will be applied. For example, if the user adds the triples `:TravelWithRulePerson prec:priority 1` and `:TravelWithRuleAnimal prec:priority 2`, the rule that is applied to a "travelsWith" edge that both starts on a person and ends on an animal will be `:TravelWithRulePerson`.

The criteria to determine which rule has the highest priority, and by consequence, for any selectors, which rule should be applied is as follows: for two given rules,

- If one rule has a priority and not the other one, the one with the explicit priority is applied. If two rules have a priority, the rule with the highest priority value is applied.
- If the two rules have the same `prec:priority` value or no `prec:priority`, the rule that has the most optional predicates is applied.
  For example a rule with two `prec:sourceLabel` (or one `prec:sourceLabel` and one `prec:destinationLabel`, or two `prec:destinationLabel`) has a higher priority than a rule with one `prec:destinationLabel` (or one `prec:sourceLabel`).
- If the two rules have the same number of optional predicates, the list of optional predicates is serialized and the first in lexical order is chosen.

Production

In Definition 21 of Section 4.1.4, we defined PREC-C contexts as functions $ctx \in Ctxc$ that map selectors to a pair composed of a template graph and the self identifier. We name this pair the *extended template*.

When a rule is applied, the chosen extended template is the one in object position of the `prec:templatedBy` triple. Its property `prec:produces` describes the template graph, as a set of template triples expressed by using RDF-star quoted triples. If a `prec:selfIs` triple exists, it will be used, else the value of `pvar:self` will be computed by using the *deduceSelf* function defined in Definition 22.

The semantics of all placeholders but `pvar:nodeLabelIRI`, `pvar:propertyIRI` and `pvar:edgeIRI` has been described by the β function in Algorithm 2. These three placeholders will be replaced by the values of `prec:nodeLabelIRI`, `prec:propertyIRI` and `prec:edgeIRI` respectively. This leverages the mechanism of predicate substitution which will be presented in Section 4.2.2.

Example 13

Listing 4.5: An example of a template graph.

```
_:MyTemplate prec:produces
    << ex:thisdataset ex:generatedBy prec: >> ,
    << pvar:self rdf:type ex:Element >> .
```

In Listing 4.5, the pair `_:MyTemplate prec:produces` has for objects two templates triples: `<< ex:thisdataset ex:generatedBy prec: >>` and `<< pvar:self rdf:type ex:Element >>`.

When `_:MyTemplate` is used, the triple `ex:thisdataset ex:generatedBy prec:` will always be produced as it contains no placeholder. The other template triple, `pvar:self rdf:type ex:Element` contains the placeholder `pvar:self`: triggering this rule will produce a triple whose subject will be the NEP itself, whose predicate is `rdf:type` and whose object is `ex:Element`.

Fallback templates

When the engine has to convert a NEP, it has to choose a rule according to the selectors computed from the NEP.

To do so, it looks for all the rules that select the NEP label or key.

Multiple cases can occur:

- If only one rule matches, the rule is applied.

- If multiple rules match, the rule to apply is decided according to the criteria described earlier in Section 4.2.1.
- There are no rules for the selector.

If there are no rules for the selector, the default behavior is to use the corresponding PREC-0 template graph described by Algorithm 3 through the *p0NodeLabelRules*, *p0EdgeRules* and *p0PropertyRules* functions.

However, users may want to customize this behavior to not use the PREC-0 fallback template graphs, for example to represent edges with an RDF triple by default instead of using the standard RDF reification.

This is performed by adding to the context a triple in the form
`S prec:templatedBy ExtendedTemplate` where `ExtendedTemplate` is the name of the extended template to use, and `S` is one of the 6 following IRIs:

- `prec:Nodes` is the fallback used for any node selector that is concerned by no rules.
- `prec:Edges` is the fallback used for any edge selector that is concerned by no edges.
- `prec:NodeProperties` is the fallback used for any property selector that is concerned by no rule and whose holder kind is "*node*", *i.e.* selectors on node properties.
- `prec:EdgeProperties` is the fallback used for any property selector that is concerned by no rule and whose holder kind is "*edge*", *i.e.* selectors on edge properties.
- `prec:MetaProperties` is the fallback used for any property selector that is concerned by no rule and whose holder kind is "*property*", *i.e.* selectors on meta-properties.
- `prec:Properties` is the fallback used for any property selector that is concerned by no rule and if the corresponding holder fallback is not specified.

4.2.2 Substitution predicates: re-using existing templates

Starting from the algorithms proposed during the second iteration presented by Section 4.1.3, the user has to provide the *precc* function with a context, a mapping from all selectors to template graphs. Building new templates for each selector might be a tedious task in practice, especially when several templates may share similar structures. To lighten the burden of defining new templates, this section proposes a solution to use existing templates and modify them: *substitution predicates*.

Definition 23 [Substitution predicate]
A *substitution predicate* allows users to specify custom substitutions to be applied to template graphs.

When the substitution predicate is used on a rule, the value of the predicate will replace all occurrences of its target in the template graph of the rule.

The syntax is as follows:

```
prec:nodeLabelIRI a prec:SubstitutionPredicate .
prec:nodeLabelIRI prec:substitutionTarget pvar:nodeLabelIRI .

:MyTemplate
   prec:produces << pvar:self rdf:type pvar:nodeLabelIRI >> .
   prec:nodeLabelIRI :knows .
```

In this example, when the substitution predicate `prec:nodeLabelIRI` is used, it will look for all `pvar:nodeLabelIRI` occurrences and replace it. For instance, in `:MyTemplate`, `pvar:nodeLabelIRI` will be replaced with `:knows`.

Five internal placeholders were introduced in Definition 15 and Definition 18:

- The internal placeholder ?*nodeLabelIRI* is the target of the built-in substitution predicate *prec*:*nodeLabelIRI*: when writing a PREC-C context, instead of explicitly defining a separate template for each node label, the writer of the context can use the same template graph, and apply to it the substitution predicate to choose the IRI. The PREC-C engine will then produce the appropriate template graph from the base template graph and the substitution predicates.
- The two internal placeholders ?*edgeIRI* and ?*propertyIRI* play the same role: they are respectively targeted by the built-in substitution predicates *prec*:*edgeIRI* and *prec*:*propertyIRI*.
- The last two internal placeholders are ?*key* and ?*label*. The PREC-C engine replaces them respectively with the property key as a string and the label as a string[6].

Remark 7 [The forged IRI of the PREC-0 functions is a substitution]
In the implementation, in the functions *p0NodeLabelRules*, *p0EdgeRules* and *p0PropertyRules*, instead of directly forging an IRI for the label/property key, the internal placeholders ?*nodeLabelIRI*, ?*edgeIRI* and ?*propertyIRI* are used in the template graph. When the template graph is used, if the corresponding substitution predicate is used, then the placeholder will be replaced using the substituted term. If the corresponding substitution predicate is not used, then the internal placeholder will be replaced with the forged IRI, as if the substitution predicate was implicitly used to replace the target placeholder with the forged IRI.

In the implementation, when a PREC-C rule is read, the template graph is modified according to the different substitution predicates that are found. The substitution predicates can be both applied directly on the template like in Definition 23 or on a PREC-C rule like in Listing 4.6.

The latter listing provides an example where a rule `:Ownership` is created: this rule matches all edge selectors that have the "Owns" label. The template graph produced from these selectors is the `prec:RDFReification` template graph: a built-in template graph designed to model edges with the classic RDF reification. However, thanks to the usage of the three substitution predicates `prec:subject`, `prec:predicate` and `prec:object`, the template graph is transformed to use the n-ary relation pattern.

Listing 4.6: Example of usage of a PREC-C context with a substitution predicate

```
## These triples are built into the PREC-C implementation

    # Built-in template graph
    prec:RDFReification
      prec:produces
        << pvar:self rdf:subject pvar:source >> ,
        << pvar:self rdf:predicate pvar:edgeIRI >> ,
        << pvar:self rdf:object pvar:destination >> ;
    prec:selfIs pvar:self .

    # Built-in substitution predicate
      # prec:edgeIRI is a substitution predicate for pvar:edgeIRI
    prec:edgeIRI a prec:SubstitutionPredicate ; prec:substitutionTarget pvar:edgeIRI .
      # prec:subject is a substitution predicate for rdf:subject
    prec:subject a prec:SubstitutionPredicate ; prec:substitutionTarget rdf:subject .
      # prec:predicate is a substitution predicate for rdf:subject
```

[6] In a sense, they can be seen as terms substituted by the substitution target *prec*:*label* and *prec*:*propertyKey*. However, these values are hard coded in the implementation, rather than the *prec*:*label* and *prec*:*propertyKey* predicates being defined as substitution predicates.

```
    prec:predicate a prec:SubstitutionPredicate ; prec:substitutionTarget rdf:predicate .
      # prec:object is a substitution predicate for rdf:subject
    prec:object a prec:SubstitutionPredicate ; prec:substitutionTarget rdf:object .

## The context explicitly provided by the user

    :Ownership a prec:EdgeLabelRule ;
      prec:label "Owns" ;
     # The following triple is not stricly required, as the fallback
     # template graphs for edges (from PREC-0) is exactly the one in
     # prec:RDFReification
      prec:templatedBy prec:RDFReification ;
      # Provide a type to the N-ary relationship
      # Substitute rdf:predicate with rdf:type
      prec:predicate rdf:type ;
      # Substitute prec:edgeIRI with :Owns
      prec:edgeIRI :Ownership ;
      # Properly name the link with the two other entities
      # Substitute rdf:subject with :owner
      prec:subject :owner ;
      # Substitute rdf:object with :owned
      prec:object :owned .

## Possible output
    _:ownership1 rdf:type :Ownership ;
        :owner _:tintin ;
        :owned _:snowy .
```

4.2.3 PREC-0 provides a PG model

Implementation wise, the PREC-C conversion is not directly applied to the source PG. As there exist many Property Graph models, and many property graph API, implementing the conversion for each supported PG API would be costly.

Instead, as mentioned in Chapter 3, the engine performs a two-step conversion:

- The first step consists in connecting to the PG API. The current implementation supports Gremlin and Cypher, reading a PG and storing it with an intermediate representation, common for both APIs. This step does not use the PREC-C context at all.
- The second step consists in converting the PGs from the intermediate representation into RDF, using the PREC-C context.

Consider the PREC-0 schema in Figure 4.2. It is an ideal candidate for this intermediate format:

- It is expressed in RDF and the final result will be in RDF. Using an RDF representation avoids choosing another representation model. In particular, it enables to define the conversion operated by an empty PREC-C context as the identity operation.
- As PREC-0 is reversible, it can be considered as "yet another Property Graph model" whose goal is to be able to store the data of any PG models we are aware of.
- As the nodes, edges and properties of the PREC-0 graph can be blank nodes, converting any PG to the PREC-0 graph corresponds to the initial step of the conversion: build an isomorphic BPG to the one to convert, or "for each NEP, choose a blank node and stick to it for the rest of the conversion".

4.3 Discussion

4.3.1 PREC-C encompasses existing conversions

In terms of PG-to-RDF graph conversion, PREC provides the same options as the two existing tools.

Neosemantics Reproducing the behavior of Neosemantics to convert Neo4j graphs to RDF is trivial. This can be achieved by using the context in Listing 4.7, *i.e.* modeling every property with the `prec:DirectTriples` template and every edge with the `prec:RdfStarUnique` template.

Listing 4.7: The PREC-C context that produces a NeoSemantics-like RDF graph

```
# Explicit triples
prec:Edges prec:templatedBy prec:DirectTriples .
prec:Properties prec:templatedBy prec:RdfStarUnique .

# The following triples are automatically added by the PREC-C engine

## Edges are modelled using RDF triples and quoted triples for properties
prec:RdfStarUnique a prec:EdgeTemplate ;
  prec:edgeIs << pvar:source pvar:edgeIRI pvar:destination >> ;
  prec:produces
    << pvar:source pvar:edgeIRI pvar:destination >> ,
    << << pvar:source pvar:edgeIRI pvar:destination >> a pgo:Edge >> .

## Properties are modelled using RDF triples and quoted triples for meta properties
prec:DirectTriples a prec:PropertyTemplate ;
  prec:entityIs << pvar:entity pvar:propertyKey pvar:propertyValue >> ;
  prec:produces << pvar:entity pvar:propertyKey pvar:propertyValue >> ;
```

The Property Graph Ontology PREC-C is also able to produce graphs that follow the PG Ontology [14]. To do so, the user needs to provide a context with explicit templates that describes how nodes, edges and properties are represented.

The ontology was described earlier by Figure 4.3. In this ontology:

- Nodes and edges are linked to their label directly represented by the label as a literal
- Nodes and edges are linked to their property through their respective predicate, `pgo:hasNodeProperty` or `pgo:hasEdgeProperty`.
- Properties have two out-coming predicate: `pgo:key` and `pgo:value` respectively linked to the property key as a literal and the property value as a literal.

Listing 4.8 describes the context that must be provided to the PREC engine.

Listing 4.8: The PREC-C context that produces a graph that complies with the PGO ontology

```
# -- Property Graph Ontology replication

prec:NodeLabels prec:templatedBy ex:pgoNodeLabel .

ex:pgoNodeLabel
    prec:produces << pvar:self pgo:label pvar:label >> .

prec:Edges prec:templatedBy ex:pgoEdge .

ex:pgoEdge
    prec:produces
        << pvar:self pgo:startNode pvar:source >> ,
        << pvar:self pgo:endNode pvar:destination >> ,
        << pvar:self pgo:label pvar:label >> ;
    prec:edgeIs pvar:self .
```

```
# -- Properties

prec:NodeProperties prec:templatedBy ex:pgoNodeProperty .
ex:pgoNodeProperty ;
    prec:produces
        << pvar:holder pgo:hasNodeProperty pvar:self >> ,
        << pvar:self pgo:key pvar:label >> ,
        << pvar:self pgo:value pvar:value >> ,
        << pvar:self a pgo:Property >> ;
    prec:entityIs pvar:self .


prec:EdgeProperties prec:templatedBy ex:pgoEdgeProperty .
ex:pgoEdgeProperty
    prec:produces
        << pvar:holder pgo:hasEdgeProperty pvar:self >> ,
        << pvar:self pgo:key pvar:label >> ,
        << pvar:self pgo:value pvar:value >> ,
        << pvar:self a pgo:Property >> ,
    prec:entityIs pvar:self .


# Meta properties are not supported by PGO: produce the empty graph
prec:MetaProperties prec:templatedBy ex:pgoMetaProperty .
ex:pgoMetaProperty
    # No value for ex:pgoMetaProperty prec:produces
    # = the empty template graph is associated to ex:pgoMetaProperty
    prec:entityIs pvar:self .

# ---- Producing the RDF node for the PG itself
# We extend the pgoEdge and pgoNodeLabel templates to add a triple between
# the pg instance and the node

# PREC is unable to apply a rule on unlabeled nodes, neither can it select
# isolated nodes only. In the current state, we produce a triple between the
# PG and the node for each node that is either labeled or is connected to another
# node (or both).

ex:pgoEdge prec:produces
    << _:thisPG pgo:hasEdge pvar:edge >> ,
    << _:thisPG pgo:hasNode pvar:source >> ,
    << _:thisPG pgo:hasNode pvar:destination >> ,
    << _:thisPG rdf:type pgo:PropertyGraph >> .

ex:pgoNodeLabel prec:produces
    << _:thisPG pgo:hasNode pvar:node >> ,
    << _:thisPG rdf:type pgo:PropertyGraph >> .

ex:pgoNodeProperty prec:produces << _:thisPG rdf:type pgo:PropertyGraph >> .
ex:pgoEdgeProperty prec:produces << _:thisPG rdf:type pgo:PropertyGraph >> .
ex:pgoMetaProperty prec:produces << _:thisPG rdf:type pgo:PropertyGraph >> .
```

The two first rules, `ex:pgoEdge` and `ex:pgoNodeLabel` are very straightforward as they simply describe the triples to produce for edges and nodes. These two templates are used as the fallback template to use for edges (through `prec:Edges`) and nodes (through `prec:NodeLabels`). Similarly, three templates are defined for node properties, edge properties and meta properties. As the predicates `pgo:hasNodeProperty` and `pgo:hasEdgeProperty` are part of PGO and specialized respectively for node properties and edge properties, we use them for the related kinds of elements through `prec:NodeProperties` and `prec:EdgeProperties`. For meta properties, as they are not supported by PGO, the empty template graph is associated with `prec:MetaProperties`.

The final set of rules is a hack to add the fact that in the Property Graph Ontology, an RDF resource is supposed to be created to represent the PG itself. For this purpose, we add extra triples to the template graphs used for nodes and edges to create a resource for the PG and link it to the current node or edge. A similar process is used for the property templates,

creating a triple isolated from the rest of the template graph. Note that because PREC-C does not support nodes with no labels and no properties (they can not be selected), it is unable to successfully emulate PGO for PGs that contain them. However, such PG can be considered to have a very limited usefulness.

4.3.2 Usability discussion

While PREC-C seems to offer a lot of possibilities thanks to its selector system, when designing the examples, we were quickly faced with the difficulty to use PREC-C to build simple mappings.

The difficulty to write PREC-C contexts comes from several aspects:

- The context written by the user is not the one used by the engine: implicit triples are added, for example the built-in templates or the representation of unspecified selectors. While this enables to have shorter context files and avoid reinventing the template of common patterns, for example the RDF reification, it forces the user to keep in mind all the implicit triples.
- A lot of different terms are defined by the ontology, from the built-in templates, the built-in substitution predicates, the predicates related to the rules... The diversity of existing terms forces the user to use the documentation to write a context. Even the author, the creator of PREC-C and its ontology, had to refer a lot to the specification and to other examples to write the example in this chapter.
- While the implicit priority system is deterministic, it may be hard to know which rule is applied in case multiple rules apply without looking at the specification. Specifying a priority for all rules may be a tedious task.

Through the course of the thesis, multiple papers [55, 50] cited PREC-C (at the time PREC) as an existing tool for RDF to PG conversion. However, in their quick summary, most of them miss some features of PREC-C, either by telling that a feature is not supported despite the feature being actually supported, or by asserting that the scope of PREC-C is narrower than what it actually is. People misunderstanding PREC-C shows that either the tool is hard to explain (or badly explained), hard to understand, or a mix of both. All tools require a time for the user to learn them. When they are facing a problem, a potential user will weight the time required to learn tools to solve the problem against the possibility to solve it by hand. In this case, all PG to RDF tools, including PREC-C, compete with an ad-hoc conversion. A tool that is too hard to use or understand does not fit its purpose, which seems to be the case of PREC-C.

Finally, in its design, PREC-C has a very schema-less approach: a PREC-C context is able to select anything and to convert any supported PG through its fallback mechanism. However, in many cases, PGs are constrained by some kind of schema, and any data not complying with these schemas would be rejected or considered meaningless. A conversion that leverages the declared or inferred schema of the PG would be a solution to both make easier the writing of contexts and increase the perceived reliability of the tool.

4.4 Conclusion

The PREC-C engine lets the user convert any PG, either through the Cypher API or the Gremlin API, in a user-configured manner through the use of a *PREC-C context*. PREC-C contexts are expressed as a list of rules. Each rule selects a group of NEPs, and lets the user choose how to represent them in RDF.

We introduced the notion of *selectors*: a PREC-C context maps all selectors to (extended) template graphs, and the PREC-C algorithm is in charge of producing an RDF graph from the template graphs related to the selectors of all NEPs of the PG. NEP selectors are expressed not only as the kind of the selected NEP and its label/property key, but also contain information about the NEPs related to the selected NEP; *i.e.* the source and destination for edge selectors, the kind and the label of the holder for property selectors. Such extra information in selectors enables to use different selectors, and therefore different template graphs, for NEPs that share the same label; for example to use a different IRI for the name of a person and the name of an animal.

As a PREC-C context can be used on any PG, the user can write their PREC-C context incrementally: starting from the empty PREC-C context, they can add rules to change the content of the produced PG until they are satisfied by the output.

The *PREC-C* rule-set quickly shows its limits for several reasons:

- The multiplication of predicates, making it hard to write a context without heavily relying on the specification. Presenting a simple running example is hard as it either requires to have a lot of implicit templates, or a very lengthy context.
- A very nested rule-set, with lots of objects and default rules. Users may feel there is a steep learning curve to learn the rules and not feel in control of the output of the engine without a high expertise.
- In particular, the PREC-C ruleset forces an approach where all NEPs are considered separately. However, properties are forced to be on a holder, and the semantics of the property may vary depending on the holder type. For example the name of a company is not the same as the name of a person. When writing a context, the natural approach consists in looking at an element, node or edge, see the missing rules in terms of node or edge label and the held properties and then move on to the next element. However, the rule-set forces the user to write separate rules for the element and its properties, with the risk of adding a new property rule for a property key that already exists. If a rule for the same property key exists and if the chosen predicate or model is different, the user would have to check and edit previous property rules to avoid conflicts by adding extra constraints on the holder.
- There is little to no data checking of the source PG, as there is a default behavior for everything. While this could be circumvented by adding some specific predicates that disable default templating, the definition of property rules would still render the mapping very permissive.

All these insights motivates the development of a simpler rule-set and with some kind of PG checking.

Chapter 5

PRSC: A higher level approach using schemas

In this chapter, we introduce a new converter named PRSC (PG to RDF Schema-based Converter).

This approach differs from the PREC-C approach presented in Chapter 3 in that, instead of building a mapping which works for all PGs, and which can be customized for certain NEPs through contexts, the contexts are going to only work for certain well-defined PGs explicitly defined by the context.

Although PG schemas are not yet standardized, it is possible to use a simple PG schema model. For example, consider the running example PG shown by Figure 5.1. A trivial schema that describes the PG is a schema that allows 1) nodes with the "Person" label and two properties: "name" and "job", 2) nodes with no label and one property: "name", and 3) edges with the "TravelsWith" label and one property: "since". As it is very likely that a schema like this will be supported by any standardization of PG schemas, it is possible to build a mapping that maps all PGs that comply with this schema to RDF.

Formally, our notion of schema is defined from Angles' PG definition (Definition 1). The notion of type is built on the kind of the PG element (node or edge), its labels and properties. The notion of a PRSC context is defined as a mapping from the types to template graphs, and the PRSC algorithm is in charge of converting the PG elements to RDF using the PRSC context. We then study a subset of PRSC contexts, named PRSC well-behaved contexts, for which we show that the conversion is reversible: the original PG can be reconstructed from the produced RDF graph.

Implementation-wise, in Figure 3.3 that shows the architecture, the PREC-0 algorithm still serves as the abstraction for heterogeneous PG APIs. From the RDF graph produced by PREC-0, the PRSC context is applied to produce a brand new RDF graph.

The work on this chapter has been accepted by the Semantic Web Journal. The main differences between this chapter and the article are 1) the proposal of an optimization of the reversion algorithm in Section 5.6, and 2) some additional extensions are proposed in Section 5.7 while the article only proposed the edge-unique extension.

5.1 PRSC in practice

The Property Graph exposed on Figure 5.1 describes the relationship between Tintin and Snowy. It is composed of two nodes. The first one holds the label "Person" and two properties: its name is "Tintin" and its job is "Reporter". The other node only has one property: its name

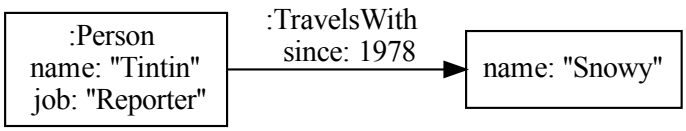


Figure 5.1: A small PG about Tintin that serves as a running example in this chapter

Listing 5.1: 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" .
```

"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 5.1. Most information that was in the PG is represented by the triples in lines 1-4 and 6. The information about since when Tintin travels with Snowy is represented through a nested RDF-star triple.

Using a user-defined mapping, PRSC is able to convert the PG in Figure 5.1 into the RDF-star graph in Listing 5.1, and more generally any Property Graph complying with the schema described in the introduction into the corresponding RDF graph. The user-defined mapping, named *PRSC context*, that the user must provide is exposed on Listing 5.2. A PRSC context is a list of rules that 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.

In particular, the PRSC context exposed on Listing 5.2 reads as follows:

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

Listing 5.2: The PRSC context that maps the PG running example to the RDF graph running example

```
_:PersonRule
  # Target: all nodes with label "Person" and two properties "name" and "job"
      a prec:PRSCNodeRule ;
      prec:label "Person" ;
      prec:propertyKey "name", "job" ;
  # Production part of the rule: a template graph
      prec:produces
        << pvar:self rdf:type      ex:Person >> ,
        << pvar:self foaf:name     "name"^^prec:valueOf >> ,
        << pvar:self ex:profession "job"^^prec:valueOf >> .

_:NamedRule
  # Target: all nodes with no label and one property "name"
      a prec:PRSCNodeRule ;
      prec:propertyKey "name" ;
  # Production part of the rule
      prec:produces
        << pvar:self foaf:name "name"^^prec:valueOf >> .

_:TravelsWithRule
  # Target: all edges with the label "TravelsWith" and one property "since"
      a prec:PRSCEdgeRule ;
      prec:label "TravelsWith" ;
      prec:propertyKey "since" ;
  # Production part of the rule
      prec:produces
        << pvar:source ex:isTeammateOf pvar:destination >>  ;
     << << pvar:source ex:isTeammateOf pvar:destination >> ex:since "since"^^prec:valueOf >> .
```

  - ⋆ Another triple with the same blank node as above 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 is used on the subject position of the triples created by this triple, `ex:since` is used as the predicate and the value of the "since" property is used as the object. In our example, the triple << `_:n1 ex:isTeammateOf _:n2` >> `ex:since 1978` is produced.
- Note that in this example, `pvar:self` is not used in lines 27 and 28. If it was used, it would be mapped to a blank node that identifies the edge. The consequence of not using it is a smaller RDF graph, at the cost of losing information if several edges of that type exist in the PG between the same nodes.

5.2 Used Property Graph formalism

Note that unlike PREC-C that enables to write rules for each kind of NEP (node, edges and properties), PRSC only allows the user to write rules about PG elements (nodes and edges). This choice of excluding rules about properties is inspired by the works of the PGSWG (Property Graphs needs a Schema Working Group), in particular by their article on PG schemas [57] that only defines node types and edges types, and no property type. Properties are considered to be part of the node and edge types, and meta properties are not considered. Following the example of the PGSWG, PRSC is therefore defined on Angles' widely used definition of PGs, rather than our more general definition of Gremlinable PGs. Extending PRSC to GPGs would raise additional challenges (especially the reversibility properties described in Section 5.5) and is left for future work.

As mentioned in its definition, the set of all PGs following Angles' definition is denoted APG. Following the motivations exposed by Section 3.6.2, the set of PGs supported by the algorithms is the set of "Blank Node Angles Property Graphs", *i.e.* Property Graphs that follow Angles' definitions and for which all nodes and edges are blank nodes. Following the convention defined by Definition 13, and as defined by Remark 6, this set is denoted $BAPG$.

Example 14 [Running example of a Property Graph (Reminder])
We remind the possible formal definition of the PG in Figure 5.1 that was provided in Example 1 from Chapter 3, denoted TT with

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

Definition 24 [The empty PG]
The empty PG, which is the PG that contains no nodes and no edges, is formalized as follows: $pg_\emptyset$ with $N_{pg_\emptyset} = E_{pg_\emptyset} = \emptyset$, $src_{pg_\emptyset} = dest_{pg_\emptyset} = labels_{pg_\emptyset} = \emptyset \to \emptyset$ and $properties_{pg_\emptyset} : \emptyset \times \emptyset \to \emptyset$.

5.3 General definitions

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

5.3.1 Domain and image of a function

Definition 25 [Domain and image of a function]
For all partial functions $f : D \to A$, 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 15
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.

5.3.2 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-tuples, S can be seen as a function iff (if and only if) $\forall (x, y_1, y_2), (x, y_1) \in S \wedge (x, y_2) \in S \Rightarrow y_1 = y_2$.

Example 16
Consider the three functions f_1, f_2, f_3 exposed in Table 5.1.

Table 5.1: 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:

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

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 8
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 26 [Functions compatibility]
Two functions f and g are compatible iff $f \cup g$ is a function, *i.e.* $\forall (x, y_f, y_g), (x, y_f) \in f \wedge (x, y_g) \in g \Rightarrow y_f = y_g$.

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

5.4 PRSC: mapping PGs to RDF graphs

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

5.4.1 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 27 [Property keys of an element]
We recall that in Definition 1, properties are described as key-value pairs.

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

Definition 28 [Type of a PG element]
A type is a triple composed of 1) the *kind* of the PG element, *i.e.* if it is a node or an edge, 2) a set of labels and 3) a set of property keys. The set of all types is denoted *Types* and is defined as $Types = (\{\text{“}node\text{”}, \text{“}edge\text{”}\} \times 2^{Str} \times 2^{Str})$.

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

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

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

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

Example 17
Table 5.2 shows the types of the PG elements in the running example.

Table 5.2: The types of the elements in the PG BTT

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

Remark 9
If two PGs pg and pg' are isomorphic, their elements share the same type.

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

5.4.2 Placeholders

Similarly to PREC-C, PRSC resorts on a mechanism of templating. As the notion of template triples has already been defined in Definition 11, we only need to define which placeholders are used by PRSC.

Definition 29 [PRSC Placeholders]
Let *valueOf* be an element that is not included in any of the sets I, B and L. In the implementation, *valueOf* is mapped to the datatype `prec:valueOf`.

The sets of PRSC placeholders are as follows:

- The node placeholders, *i.e.* the members of the set PN, are ?$self$, ?*source* and ?*destination*. They are respectively placeholders for the NEP itself, the source node of an edge and the destination node of an edge
- There is no IRI placeholder, *i.e.* the set PI is empty.
- The set of literal placeholders is $PL = Str \times \{valueOf\}$. Elements of PL can be noted using the same notations as for the literals, for example (“*name*”, *valueOf*) and “*name*”^{valueOf} denote the same literal placeholder. Elements of PL serve as placeholder to be replaced with an RDF literal that represents the value of a property in the PG.

5.4.3 PRSC context

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

Definition 30 [PRSC Context]
A PRSC context $ctx : Types \rightarrow 2^{Templates}$ is a partial function that maps types to template graphs.

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

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

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

The set of all context functions is denoted Ctx.

Definition 31 [Complete PRSC contexts for a given PG]
A PRSC context is said complete for a Property Graph $pg \in BPGs$ iff there is a template graph defined for each type used in pg. The set of all complete contexts for a PG pg is noted $Ctx_{pg} = \{ctx \in Ctx \mid \forall m \in N_{pg} \cup E_{pg}, typeof_{pg}(m) \in Dom(ctx)\}$.

Remark 10
Note that the type system described in Definition 28 is trivial to resolve as the type of a PG element m, denoted by $typeof_{pg}(m)$, only depends on its kind (node or edge), the list of its labels and the list of its property keys. For this reason, checking if a PRSC context ctx is complete for a Property Graph pg is also trivial as it consists in computing the type of each PG element of pg and checking if all types are in the domain of ctx.

Example 18
Table 5.3 exposes an example of a complete ctx context function for our running example. First, the function is a context as all template graphs are valid: The placeholders $\text{“}name\text{”}^{valueOf}$ and $\text{“}job\text{”}^{valueOf}$ are only used in types with the associated property key. The fact that the property key “*since*” in the third type has no associated placeholder occurrence in the template graph does not invalidate the context. The placeholders $?source$ and $?destination$ are only used in the third type, which is an edge type. Then, all three types used by our running example have an associated template graph, so it is a complete context for the PG exposed in Example 5.1.

Table 5.3: 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 19
The function *ctx* in Table 5.4 is not complete for the PG *BTT* as its domain lacks the type of _:*n2* and the type of _:*e1*.

Table 5.4: An incomplete context for the Tintin PG

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

Example 20
The function *ctx* below in Table 5.5 is not a context because “*surname*”, 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 key in $\{\text{“}name\text{”}, \text{“}job\text{”}\}$.

Table 5.5: A function that is not a context

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

5.4.4 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 32 [Property value conversion]
For the conversion of property values to literals, we consider that we have a fixed total injective function $toLiteral : V \to L$, common for all PGs and contexts. We suppose that *toLiteral* is reversible, *i.e.* we are able to compute $toLiteral^{-1}$.

The exact definition of *toLiteral* obviously depends on the specific set V, which in turns depends on the PG implementation. Defining it is however relatively straightforward for the most common datatypes, by using the standard XML Schema Datatypes [76] or the `rdf:JSON` datatype [77].

Definition 33 [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 graphs built by converting all elements of the PG, into RDF. The conversion of a single element is performed by the *build* function.

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

$$\beta_{pg,m} : \begin{cases} Templates & \rightarrow RdfTriples \\ PL \cup L & \rightarrow L \\ I & \rightarrow I \\ PN & \rightarrow B \end{cases}$$

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

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

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

Example 21
Table 5.6 exposes the resolution of *prsc* on the running example.

Table 5.6: Application of a PRSC context on the running example

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

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

Algorithm 8: The *prsc* function

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

Main Function *prsc(pg, ctx)*:
  $rdf \leftarrow \{\}$
  forall *PG element* $m \in N_{pg} \cup E_{pg}$ **do**
    $tps \leftarrow ctx(typeof_{pg}(m))$
    /* *build* function */
    $built \leftarrow \{\}$
    forall $tp \in tps$ **do**
      $built \leftarrow built \cup \{\beta(tp, pg, m)\}$
    $rdf \leftarrow rdf \cup built$
  return *rdf*

/* In the formal definition, pg and m are implicitly passed to β */
Function $\beta(tp, pg, m)$:
  if $tp \in Templates$ **then**
    $(s, p, o) \leftarrow tp$
    return $(\beta(s, pg, m), \beta(p, pg, m), \beta(o, pg, m))$
  else if $tp \in L$ **then return** tp
  else if $tp \in I$ **then return** tp
  else if $tp \in PL$ **then**
    $(key, valueOf) \leftarrow tp$
    return $toLiteral(properties_{pg}(m, key))$
  else
    $assert(tp \in PN)$
    switch tp **do**
      case $?self$ **do return** m
      case $?source$ **do return** $src_{pg}(m)$
      case $?destination$ **do return** $dest_{pg}(m)$

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

Algorithm 8 gives an algorithmic view of the *prsc* function presented by Definition 33.

5.4.5 Complexity analysis

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

Functions considered constant

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

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

Evaluating if something is a member of a given set, for example if an entity is a member of the set N_{pg}, is generally considered to be in constant time thanks to hash maps[1].

Considered metrics

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

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

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

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

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

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

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

- The number of types supported by the context.

$$NbTypes = |Dom(ctx)|$$

In RDF-star, quoted triples can be used as subject or object of other triples, without limit on how deeply triples can be nested. In practice, however, it is rare to have more than one level of nesting. Usually, users are expected to use atomic RDF triples like `_:tintin :travelsWith _:haddock` or to use RDF-star triples with a depth of one like `<< _:tintin :travelsWith _:haddock >> :since 1978`. We therefore consider the *depth* of any triple to be bound by a constant. As a consequence, in all functions processing terms and triples recursively (such as β in Algorithm 8), we can ignore the recursion depth in the complexity analysis.

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

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

Complexity of *ctx* calls in the *prsc* function

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

- The type of m in the PG pg must be computed. $typeof_{pg}(m)$ has a complexity of $\mathcal{O}(1)$:
  - Evaluating if m is a node or an edge is constant as checking if an element is a member of a set is constant.
  - Calls to $labels_{pg}(m)$ and $keys_{pg}(m)$ are considered constants in Section 5.4.5.
- In the complexity analysis, we consider that *ctx* is implemented as a hash map from the types to the template graphs. A *ctx* call would first need to compute the hash of the type. To do so, it has to look at all the labels and properties in the type in a deterministic order; for this, the labels and keys need to be sorted, which has a complexity of $\mathcal{O}(TypeComplexity * \ln(TypeComplexity))$. After the hash has been computed, the cost of retrieving the template graph has an amortized constant complexity. The overall complexity of a *ctx* call is $\mathcal{O}(TypeComplexity * \ln(TypeComplexity))$.

Complexity of *prsc*

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

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

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

5.5 PRSC reversibility

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

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

5.5.1 The notion of reversibility

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

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

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

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

Example 22 [A trivially non-reversible context]
Consider the context $ctx_\emptyset$ such that for all types, it returns the empty template graph, *i.e.* $\forall type \in Types, ctx_\emptyset(type) = \emptyset$. As it is complete for all Property Graphs, it is possible to use this context on any Property Graph. However, applying the context $ctx_\emptyset$ produces the empty RDF graph. Therefore, the use of the context $ctx_\emptyset$ makes the function $prsc$ not injective, and therefore not reversible.

Example 23 [A more realistic example of a non-reversible context]
Another example of a non-reversible context is the context exposed in Table 5.3: while this context can be applied on PGs in which edges have the "since" property, the value of this property will never appear in the produced RDF graph.

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

5.5.2 Well-behaved contexts

Characterization function

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

Definition 34 [Characterization function]

The κ function maps:

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

$$\kappa : \begin{cases} \mathit{Templates} \cup \mathit{RdfTriples} & \rightarrow 2^{\mathit{RdfTriples}} \\ L \cup PL & \rightarrow \{L\} \\ I & \rightarrow 2^{I} \\ B \cup PN & \rightarrow \{B\} \end{cases}$$

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

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

Example 24 [κ applied to the running example from Figure 5.1]

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

Table 5.7 provides an example of applying κ on the running example context of Table 5.3.

Remark 11 [κ on terms and triples is, as expected, a super-set of the possible generated values]
When comparing the definition of the κ function with the β functions defined in Section 5.4.4, it appears that:

- For elements in B, PN, L and PL, 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 12 [The result of *build* is, as expected, a subset of the result of κ]
The *build* function, from which *prsc* is defined, uses β on each template triple. After β is applied, the union of the singletons containing each triple is computed. This is similar to the definition of κ on a set of triples.

From Remark 11, 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 13 [A template and its produced values share the same image through κ]
When using the κ function, elements in B and PN both map to B, and elements in L and PL 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 PN map for all PGs $pg \in BAPG$ to elements of N_{pg} and E_{pg}, which are both subsets of B.
- Elements in PL map to elements in $Img(toLiteral)$, which is a subset of L.
- Elements in L and I are mapped to themselves.
- Elements in *Templates* apply the β function recursively on their members.

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

As mentioned previously, the role of κ is to allow us to determine whether two template triples with placeholders may produce the same triple. It maps all placeholders to a super-set [2] of all elements they can generate with the *build* function. All RDF triples are mapped by the κ function to a subset of *RdfTriples* they are a member of.

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

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

[2]Note that as κ maps to a super set, it may catch false positives. For example, PL can only generate elements in $Img(toLiteral)$, but the κ function considers that all elements of L can be generated from PL. For the scope of this thesis, 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.

Proof. Per the Definition 33 of *build*, an RDF triple can only be generated by a template graph by the application of β to one of its template triples. Per Remark 13, the generated triple and the corresponding template triple have the same image through κ. □

Definition 35 [*unique* template triple]
A template triple tp is *unique* in a set of template triples if no other template triple in the set has the same image through κ as tp.

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

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

Combined with Remark 13, what $unique(tp, tps)$ tells us is that any triple, with the same image through κ as tp , can not have be generated by any other element of tps than tp itself. This leads us to Theorem 1 below.

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

$\forall pg \in ABPG, \forall ctx \in Ctx_{pg}, \forall m \in (N_{pg} \cup E_{pg})$, 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) $unique(tp, tps)$, i.e. $(\forall tp' \in tps, \kappa(tp) = \kappa(tp') \Rightarrow tp = tp')$
- (B2) $\kappa(td) = \kappa(tp)$
- (C) $td \notin build(\{tp\}, pg, b)$

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

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

□

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

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 36
(Well-behaved contexts)

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

- *Element provenance*: all generated triples must contain the blank node that identifies the node or the edge it comes from. This is achieved by using the $?self$ placeholder in all template triples:
  - $\forall tp \in tps, ?self \in tp$
- *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, nor for edges, the source and destination node. Each of these pieces of information must be present in an unambiguously recognizable triple pattern.
  - $\forall key \in keys(type), \exists tp \in tps \mid unique(tp, tps) \wedge (key, valueOf) \in tp$
  - $kind(type) =$ "$edge$" $\Leftrightarrow \exists tp \in tps \mid unique(tp, tps) \wedge ?source \in tp$
  - $kind(type) =$ "$edge$" $\Leftrightarrow \exists tp \in tps \mid unique(tp, tps) \wedge ?destination \in tp$

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

Remark 14 [Handling multiple $sign_{ctx}$ candidates]
In the case where there are multiple template triples candidates to become the signature template triple, the choice of the signature template triple among the candidates is generally not important.

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

Remark 15 [The template graphs used in well-behaved contexts are not empty]
A well-behaved context cannot map a type to an empty template graph: the *signature template triple* criterion ensures that every template graph contains at least one template triple: $\forall tps \in Img(ctx), \exists tp \in tps \Leftrightarrow |tps| \geqslant 1$.

Remark 16 [Inside a well-behaved context, all template graphs are different from all others]
For any well-behaved context *ctx*, two types cannot share the same template graph. Indeed, if two types share the same template graph, *i.e.* there are two types *type1* and *type2* with $type1 \neq type2$ such that $ctx(type1) = ctx(type2)$, it would contradict the *signature template triple* criterion as it would lead to $type1 = type2$.

Example 25
Table 5.7 studies the context used in our running example, exposed in Example 18.

Table 5.7: The running example context with the corresponding values through κ

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

- The type *tn1* matches all criteria of a well-behaved PRSC context:
  - All triples contain $?self$.
  - At least one template triple is a signature: the image through κ of $(?self, rdf{:}type, ex{:}Person)$ is not contained in the image through κ of other types. It is also the case of $(?self, ex{:}profession, \text{“}job\text{”}^{valueOf})$.
  - 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 26 [A well-behaved context for the running example]
Let ctx_{TTWB} be the function described in Table 5.8. In this new context, an arbitrary $ex{:}NamedEntity$ IRI is used to sign the PG nodes that have no label and only a name, and a classic RDF reification is used to model the PG edges.

Table 5.8: An example of a complete and well-behaved context for the Tintin Property Graph.

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

This context is well-behaved:

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

Listing 5.3 is the RDF graph produced by the application of the context ctx_{TTWB} on the PG BTT. Each part that starts with a # denotes a *build* application to the PG element described in the comment. The elements are ordered in the same order as their type in Table 5.8, and the RDF triples and the template triples that produced them are also in the same order.

Listing 5.3: 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 17 [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:

$$\forall pg \in BAPG, ctx \in {Ctx_{pg}}^{+}, |prsc(pg, ctx)| = 0 \Leftrightarrow pg = pg_{\emptyset}$$

Indeed, Remark 15 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.

Implementation and complexity analysis

PRSC well-behaved contexts will be proved to be reversible in Section 5.5.3, meaning that producing an RDF graph from them will preserve all information stored in the PG. It is therefore important to be able to determine if in practice, it is possible to compute if a PRSC context is well-behaved.

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

Proof. Consider any atomic RDF term t:

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

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

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

□

Remark 18 [Implementing κ and complexity analysis]
The function κ is defined to return sets, some of them being infinite sets. While this definition is useful to prove different theorems, it is not practical from an implementation perspective.

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

$$\kappa_{impl}(x) = \begin{cases} \{\kappa(triple) \mid triple \in x\} & \text{if } x \in RdfTriples \cup Templates \\ (\kappa_{impl}(s), \kappa_{impl}(p), \kappa_{impl}(o)) & if x = (s,p,o) \in RdfTriples \cup Templates \\ \lambda & \text{if } x \in L \cup PL \\ x & \text{if } x \in I \\ \beta & \text{if } x \in B \cup PN \end{cases}$$

Compared to Definition 34, we replaced:

- the singleton $\{x\}$ with x, in the case where $x \in I$,
- the sets L and B with two constants λ and δ that are not elements of I,

- the cross product with a simple triple of the values returned for each element of x when x is a triple.

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

- For any x that is not a triple nor a graph, calls to this function can be done in constant time, by simply checking the type of x.
- When x is a triple, calls to this function involve recursive calls up to the depth of x, which we consider to be bounded by a constant (see Section 5.4.5). So it is also done in constant time.
- When x is a graph, calls to this function involve calling κ_{impl} on each triple of the graph. As the call on a triple is constant, the call on the graph x has a linear complexity depending on the size of the graph.

Note that:

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

Remark 19 [Complexity of checking if a PRSC context is a well-behaved]
The first task to check if a context *ctx* is well-behaved consists in computing the value through κ of all triples used in it. As the depth of a template triple is considered to be negligible, the complexity is the number of template triples, bounded to the number of types multiplied by the size of the biggest template graph: $\mathcal{O}(NbTypes * BiggestTemplateSize)$.

After the value through κ of all template triples have been computed, for each type, we need to check if the type complies with the three criterion exposed in the Definition 36.

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

BiggestTemplateSize) complexity for the whole context.

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

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

5.5.3 Reversion algorithm

Algorithm 9 aims to convert an RDF graph, that was produced from a PG and a known well-behaved context, into the original PG.

It is a four steps algorithm: 1) it finds the elements of the PG, by assuming they are the same as the blank node in the RDF graph, 2) it gives a type to all PG elements with the *FindTypeOfElements* function in Algorithm 10[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 11, and 4) it looks for the source, destination and properties of all elements with the *buildpg* function in Algorithm 12.

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

Finding the elements of the PG

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

Definition 37 [List of blank nodes used in an RDF graph]
For every RDF graph *rdf*, $BNodes(rdf)$ is the set of blank nodes in *rdf* *i.e.* $\forall rdf \subseteq RdfTriples, BNodes(rdf) = \{bn \in B \mid \exists t \in rdf, bn \in t\}$.

Example 27
Let GTT be the RDF graph exposed on Listing 5.1. $BNodes(GTT) = \{\_{:}tintin, \_{:}snowy\}$

Algorithm 9: The main algorithm to convert back an RDF graph into a PG by using a context

Input: $rdf \subset RDFTriples$, $ctx \in Ctx^+$
Output: A BPG or error
1 **Main Function** *RDFToPG(rdf, ctx)*:
2 $Elements \leftarrow BNodes(rdf)$
3 $typeof \leftarrow FindTypeOfElements(rdf, ctx, Elements)$
4 $builtfrom \leftarrow AssociateTriplesWithElements(rdf, Elements, typeof)$
5 **return** $buildpg(ctx, Elements, typeof, builtfrom)$

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

Algorithm 10: Associate the elements of the future PG with their types

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

Function *FindTypeOfElements(rdf, ctx, Elements)*:
  $typeof \leftarrow \{\}$
  forall *element* $m \in Elements$ **do**
    /* Find possible types */
    $candtypes_{nodes} \leftarrow \{\}$
    $candtypes_{edges} \leftarrow \{\}$
    forall *triple* $t \in rdf \mid m \in t$ **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 11: Associate each triple to the element that has produced it

Input: $rdf \subset RDFTriples$, $Elements = BNodes(rdf)$, $typeof : Elements \mapsto Type$
Output: A mapping $Elements \rightarrow 2^{RdfTriples}$ 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
      /* No blank node in bns, or multiple PG nodes but no PG edges, or multiple PG edges */
      raise *Error(No element provenance)*
  return *builtfrom*

Algorithm 12: Produce a PG from the previous analysis of the elements and triples.

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

Output: A BPG or error

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

Function *extract*$(placeholder, tds, tps)$**:**
- $values \leftarrow \{\}$
- **forall** $tp \in tps \mid unique(tp, tps) \wedge placeholder \in tp$ **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*

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

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

Proof.

- The *build* function, described in Section 5.4.4, produces specific triples depending on the given template. The template graphs cannot contain blank nodes: the blank node

produced by *prsc* are forced to be the elements of the converted BPG. So $BNodes(rdf) \subseteq N_{pg} \cup E_{pg}$.

- From Remark 15, we know that $ctx(typeof_{pg}(m))$ contains at least one triple pattern tp. Combined with the *element provenance* criterion from Definition 36, we know that $?self \in tp$. When β is applied to tp, a triple that contains m is forced to appear, meaning that $N_{pg} \cup E_{pg} \subseteq BNodes(rdf)$.

□

Theorem 2 proves the correctness of the $Elements \leftarrow BNodes(rdf)$ step in Algorithm 9.

Finding the type related to each element

In this part of the proof, we show that the *FindTypeOfElements* function from Algorithm 10 is correct, *i.e.* it is able to find back the right type of all elements m in the original pg graph.

Lemma 3
If a data triple shares the same value through κ as one of the signature triples of a type, then the element from which the data triple was produced must be of this type:

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

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

Assuming (A) $\kappa(td) = \kappa(sign_{ctx}(type))$
$td \in build(ctx(typeof_{pg}(m)), pg, m)$
$\Rightarrow$ $\exists tp \in ctx(typeof_{pg}(m)) \mid \kappa(td) = \kappa(tp)$ [Lemma 1]
$\Rightarrow$ $\exists tp \in ctx(typeof_{pg}(m)) \mid \kappa(sign_{ctx}(type)) = \kappa(tp)$ [A]
$\Rightarrow$ $\exists tp \in ctx(typeof_{pg}(m)) \mid \kappa(sign_{ctx}(type)) = \kappa(tp) \subseteq \kappa(ctx(typeof_{pg}(m)))$ [$tp \in ctx(typeof_{pg}(m))$ and by construction of κ]
$\Rightarrow$ $typeof_{pg}(m) = type$ [Signature template triple in Definition 36]

□

Definition 38 [Formalizing *candtypes*]
For a given blank node/PG element b, $candtypes_{nodes}$ and $candtypes_{edges}$, introduced in Algorithm 10, can be formally defined as:

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

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

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

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

Table 5.9 provides an overview of the cardinality of the different *candtypes* sets.

Table 5.9: A simple view of Theorem 3

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

Proof.

$\forall b \in BNodes(rdf), \forall type \in candtypes_{nodes}(b)$
Per Definition 38, $kind(type) =$ "$node$" $\wedge\ \exists td \in rdf \mid b \in td \wedge \kappa(sign_{ctx}(type)) = \kappa(td)$.
We are going to restrict the portion of the graph rdf where such triples td may be located:

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

- We see that all triples td contributing to $candtype_{nodes}(b)$ must have been produced by the signature triple template applied to a node from the PG. Also remember that td must contain b.
- If $b \in N_{pg}$, then the signature triple of $ctx(typeof_{pg}(b))$ must have generated a td containing b (since it must contain $?self$, according to Definition 36), so $typeof_{pg}(b) \in candtype_{nodes}(b)$. Furthermore, no other node can produce a td containing b ($?self$ is the only blank node placeholder in node type templates), so $candtype_{nodes}(b)$ can not contain any other type. Therefore $candtype_{nodes}(b) = \{typeof_{pg}(b)\}$.
- If $b \in E_{pg}$, it is impossible to produce the blank node b from any node $m \in N_{pg}$ (again, $?self$ is the only blank node placeholder in node type templates). No td containing b can be found, so $candtypes_{nodes}(b)$ is empty.

The reasoning for $candtypes_{edges}(b)$ when b is an edge is similar to the one for $candtypes_{nodes}(b)$ when b is a node: only b can produce triples containing itself, and it will, because having at least one signature triple with $?self$ is imposed by Definition 36. So $candtype_{edges} = \{typeof_{pg}(b)\}$.

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

Remark 20
Theorem 3 not only shows that the *FindTypeOfElements* function in Algorithm 10 will always find the right $typeof_{pg}$ function by using *candtypes*, *i.e.* that it is computable from *rdf* and *ctx*, but Table 5.9 also explicitly shows that the *Error(No type found)* scenario can not appear if the RDF graph was produced from a PG, making the *FindTypeOfElements* function both sound and complete.

Remark 21 [Using different signatures to determine the types]
As mentioned previously, the choice of the signature template triple $sign_{ctx}(type)$ of a given type is not important for the reversion algorithm. Choosing one or another signature template triple only leads to other data triples being used to determine the type of the elements. However, the end-result does not change.

Finding the generated triples for each PG element

For each PG element, given the produced RDF graph and the type of this PG element, we are able to compute the list of RDF triples produced from this PG element. In other words, Algorithm 11 correctly partitions the RDF graph into sub-graphs describing each element of the original PG.

Theorem 4
In Algorithm 11, assuming that the passed value of *typeof* is equal to $typeof_{pg}$, $\forall m \in N_{pg} \cup E_{pg}, build(ctx(typeof_{pg}(m)), pg, m) = builtfrom(m)$.

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

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

- Let us first recall that blank nodes in *rdf* can only be produced via the placeholders from *PN*: $?self$, $?source$, and $?destination$. Let us also recall that every triple pattern in the Well-behaved context *ctx* must contain $?self$ (per the *element provenance* criterion).
- If *bns* contains only one blank node m, then m must come from $?self$, and the corresponding triple pattern must then belong to $ctx(typeof(m))$. *td* must then have been produced by $build(ctx(typeof(m)), pg, m)$ so putting it in $builtfrom(m)$ is correct.
- If *bns* contains multiple blank nodes, *td* must have been produced by a template triples with several placeholders from *PN*.

  - Node template graphs can contain only one placeholder from PN: $?self$. No $m \in N_{pg}$ could then have produced td. It follows that td must have been produced by the template graph of an edge.
  - Edge template graphs can contain several placeholders from PN. But by definition of β, only $?self$ can be mapped to an edge (when $m \in E_{pg}$); $?source$ and $?destination$ are always mapped to nodes. Of the multiple blank nodes in bns, exactly one of them, m, must therefore be an edge, and come from $?self$. Following the same reasoning as above, when bns contained only one blank node, we conclude that td must then have been produced by $build(ctx(typeof(m)), pg, m)$ and that putting it in $builtfrom(m)$ is correct.

- *Error(No element provenance)* will never be raised if *rdf* was produced by *prsc*: each triple will contain at least one blank node generated from $?self$ (per the *element provenance* criterion), and if there are multiple blank nodes we showed that there must be only one edge blank node.

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

Building the PG element

Projecting Property Graphs As an RDF graph is defined as a set of RDF triples, any subset of that set, as well as the union of two RDF graphs, are formally defined and are also RDF graphs. Algorithm 12 constructs back the original PG in an iterative manner. To prove its correctness, we need operators similar to $\subset$ and $\cup$ for RDF graphs, but for our formalization of PGs.

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

Let *pg* be an APG, *i.e.* a PG that follows Angles' definition (Definition 1). Note that in this subsection, *pg* is allowed to contain no blank nodes.

Definition 39 [π projection of a Property Graph on an element]
The π projection of a PG on a node is equal to the PG with only the node itself. The π projection of a PG on an edge is the edge, and its source and destination nodes without the labels and properties of these nodes.

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

- If $m \in N_{pg}$, $N_{\pi_m(pg)} = \{m\}$, $E_{\pi_m(pg)} = \emptyset$, $src_{\pi_m(pg)} = dest_{\pi_m(pg)} = \emptyset \to \emptyset$
- If $m \in E_{pg}$, $N_{\pi_m(pg)} = \{src_{pg}(m), dest_{pg}(m)\}$, $E_{\pi_m(pg)} = \{m\}$, $src_{\pi_m(pg)} = \{m \mapsto src_{pg}(m)\}$, $dest_{\pi_m(pg)} = \{m \mapsto dest_{pg}(m)\}$
- $\forall x \in N_{\pi_m(pg)} \cup E_{\pi_m(pg)}, labels_{\pi_m(pg)}(x) = \begin{cases} labels_{pg}(x) & \text{if } x = m \\ \emptyset & \text{otherwise} \end{cases}$
- $\forall key \in keys_{pg}(m), properties_{\pi_m(pg)}(m, key) = properties_{pg}(m, key)$, all other values are undefined.

Definition 40 [Property Graph merge operator $\oplus$]
The merge operator $\oplus$ is the inverse of the projection operator π. It can only be used on two PGs that are compatible, *i.e.* (1) a PG element defined as a node is not defined as an edge in the other, (2) an edge defined in both PGs have the same source and destination in both, and (3) if in both PGs, the value of a property key on the same PG element is defined, the values should be the same. The $\oplus$ operator builds a PG with the PG elements, labels and properties of both PGs.

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

- $E_{pg'} \cap N_{pg''} = \emptyset \ \wedge \ N_{pg'} \cap E_{pg''} = \emptyset$
- $src_{pg'}$ is compatible with $src_{pg''}$ and $dest_{pg'}$ is compatible with $dest_{pg''}$ (see compatibility definition in Section 5.3.2).
- $properties_{pg'}$ is compatible with $properties_{pg''}$.

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

- $N_{pg} = N_{pg'} \cup N_{pg''}$
- $E_{pg} = E_{pg'} \cup E_{pg''}$
- $src_{pg} : E_{pg} \to N_{pg}$, $src_{pg} = src_{pg'} \cup src_{pg''}$.
- $dest_{pg} : E_{pg} \to N_{pg}$, $dest_{pg} = dest_{pg'} \cup dest_{pg''}$.
- $\forall m \in N_{pg} \cup E_{pg}, labels_{pg}(m) = \begin{cases} labels_{pg'}(m) \cup labels_{pg''}(m) & \text{if both are defined} \\ labels_{pg'}(m) & \text{if } labels_{pg'}(m) \text{ is defined} \\ labels_{pg''}(m) & \text{if } labels_{pg''}(m) \text{ is defined} \end{cases}$
- $properties_{pg} : (N_{pg} \cup E_{pg}) \times Str \to V$, $properties_{pg} = properties_{pg'} \cup properties_{pg''}$.

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

[a] The empty PG $pg_\emptyset$ is the graph such that $N_{pg_\emptyset} = E_{pg_\emptyset} = \emptyset$.

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 PG elements is equal to the PG itself:

$$\forall pg \in APG, pg = \bigoplus_{m \in N_{pg} \cup E_G} \pi_m(pg)$$

Proof. The proof is provided in Appendix B. □

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

can only add information, and in the case merging two PGs would lead to an inconsistent PG, the merge operator is undefined.

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

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

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

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

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

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

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

Completing the proof of the reversion algorithm We are now back to proving that for all well-behaved contexts *ctx*, the *RDFToPG* function presented in Algorithm 9 is an implementation of the $prsc^{-1}$ function. *pg* is a PG for which we know the value of $prsc(pg, ctx) = rdf$. We are focusing on the last line of Algorithm 9, where the *buildpg* function is invoked.

To prove the correctness of the *buildpg* function in Algorithm 12, starting from an empty PG g, we are going to show that at each iteration, we are adding to the PG g the π projection of *pg* on an element m. After iterating on all elements, as we merged the π projection of all PG elements, the PG g ends up being equal to the PG *pg* itself.

Lemma 5 [Merging the projection of one PG element to the reconstructed PG]
In Algorithm 12, assuming that the *typeof* parameter is equal to $typeof_{pg}$ and *buildfrom* is a total function that maps all PG elements b to $build(ctx(typeof_{pg}(b)), pg, b)$, at the end of an iteration of an element $b \in N_{pg} \cup E_{pg}$ after line 13, the computed PG g_{after} is equal to $g_{before} \oplus \pi_b(pg)$, where g_{before} is the PG g at the beginning of the iteration between lines 3 and 4.

Proof. The PG $\pi_b(pg)$ is described in Table 5.10. 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 5.10: Description of the PG projection that is built in Algorithm 12

	$b \in N_{pg}$	$b \in E_{pg}$
$N_{\pi_b(pg))}$	$\{b\}$	$Img(src_{\pi_b(pg)}) \cup Img(dest_{\pi_b(pg)})$
$E_{\pi_b(pg)}$	$\emptyset$	$\{b\}$
$src_{\pi_b(pg)}$	$\emptyset \to \emptyset$	$b \mapsto \boldsymbol{src_g(b)}$
$dest_{\pi_b(pg)}$	$\emptyset \to \emptyset$	$b \mapsto \boldsymbol{dest_g(b)}$
	$b \in N_{pg} \cup E_{pg}$	
$labels_{\pi_b(pg)}$	$\{b \mapsto labels(type)\}$	
$properties_{\pi_b(pg)}$	$\bigcup_{key \in keys(type)} \{(b, key) \mapsto \boldsymbol{properties_g(b, key)}\}$	

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

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

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

The last instructions differ for $?source$ / $?destination$ and P. In the case of $?source$ and $?destination$, the obtained value is directly the value of the PG node; in the case of P, the obtained RDF literal needs to be converted into the proper PG property value, which is possible because $toLiteral^{-1}$ is assumed to be computable in Section 5.4.4.

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

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

□

Remark 22 [Completeness of *buildpg*]
In the case where *rdf* is built from a PG *pg*, the value that a placeholder is mapped to is the same everywhere, so we never run at the risk of encountering multiples values, *i.e.* Error(Not exactly one value for a placeholder) is never raised. Furthermore, the proof of Lemma 5 shows that *Error(Unique data triple is not unique)* may not be raised, because we know that each unique template triple has produced one data triple.

Theorem 6 [Merging the projection of all PG elements to the reconstructed PG]
Under the same assumptions as Lemma 5, the PG returned by Algorithm 12 is the original *pg*, the PG that was used to produce the RDF graph $rdf = prsc(pg, ctx)$.

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

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

□

As *buildpg* in Algorithm 12 correctly reconstructs pg, and as its value is directly returned by the *RDFToPG* function in Algorithm 9, we have finally proven that the latter is a sound and complete implementation of $prsc^{-1}$ function for any well-behaved PRSC context ctx.

Complexity analysis

Let us now discuss the complexity of the *RDFToPG* function described in Algorithm 9. In this discussion, a new metric is considered: the number of triples in the RDF graph: $NbTriples = |rdf|$. It is assumed that we have first checked if the context ctx is a well-behaved PRSC context, and computed the $sign_{ctx}$ function so it can now be called in constant time.

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

The *FindTypeOfElements* function in Algorithm 10 uses three nested loops and only uses constant time operations: its complexity is $\mathcal{O}(NbOfPGElements * NbTriples * NbTypes)$.

Trivially, Algorithm 11 has a complexity of $\mathcal{O}(NbOfPGElements + NbTriples)$.

In Algorithm 12:

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

  In the context of the *buildpg* function, the *extract* function is always called with a template graph *tps* from the PRSC context and a sub-graph of the RDF graph *rdf* as *tds*. the complexity of the calls of the *extract* function in the *buildpg* function is $\mathcal{O}(BiggestTemplateSize^2 + NbTriples)$[4].
- The *buildpg* function loops on all PG elements.
  - Notice that while $ctx(typeof(b))$ is called multiple times, it can be called once and then its value can be cached. Its cost is $\mathcal{O}(TypeComplexity * \ln(TypeComplexity))$ as mentioned in Section 5.4.5.

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

  - There are at most $2 +$ *TypeComplexity* calls of the *extract* function. All of them have a $\mathcal{O}(BiggestTemplateSize * NbTriples)$ complexity.
  - The complexity of each iteration is $33\,\mathcal{O}(TypeComplexity * \ln(TypeComplexity) + TypeComplexity * BiggestTemplateSize^2 * NbTriples))$

- The overall complexity of the *buildpg* function is $\mathcal{O}(NbOfPGElements * TypeComplexity * (\ln(TypeComplexity) + BiggestTemplateSize^2 * NbTriples))$

The overall complexity of the *RDFToPG* function presented in Algorithm 9 presented in this section for an RDF graph produced from a PG and a well-behaved PRSC context is:

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

The *RDFToPG* function is computable in polynomial time w.r.t. all the considered metrics, and is therefore considered as tractable.

5.5.4 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 unambiguously capture the type of each resource, which is usually considered to be good practice. The type can either be explicit, through a triple with *rdf*:*type* as the predicate, or implicit through a property that is only used by this type. For example, the template graph for a type *Person* could contain a signature template triple like $(?self, :personId, \text{“}pid\text{”}^{valueOf})$. 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 [48, 14] propose to map each label to an RDF type or a literal used as the object of a specific predicate like `pgo:label`. More generally, users may want to use a composite key to sign their types. For these kinds of mappings, our approach of identifying the type by finding a single signature template is not sufficient. It requires finding all the signature template triples and deciding to which type they are associated, for example through a Formal Concept Analysis process. This could be studied as a future extension of the PRSC reversion algorithm.

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

5.6 Optimizing the reversion algorithm

While the provided *RDFToPG* function has been proved to be reversible and tractable, in practice, the exposed complexity is disappointing. The motivations behind presenting a suboptimal algorithm in Section 5.5.3 are 1) to provide an easier to understand version, and 2) to make proofs easier to follow.

In this section, we provide new versions of some of the previous algorithms to improve their complexity.

5.6.1 Checking if a context is a PRSC well-behaved context

Algorithm 13 shows an optimized method to check if a context is a PRSC well-behaved context. The main idea of this algorithm is to use hash-maps from the value through κ of template triples to a list of something that is supposed to be unique: the template triple itself for the "no value loss" criterion and the type for the "signature template triple" criterion. After exploring all template triples, we check if the template triple or the type is the only element in the list.

Ideas behind the new algorithm

"No value loss" criterion The "no value loss" criterion check is implemented by 1) building a map with the value through κ as the key and the list of template triples with this value through κ as the value, 2) listing the list of placeholders that must be found for the considered type and 3) for all template triples that are alone in their list, *i.e.* the ones that are unique, looking for each placeholder in the template triples and then remove this list from the list of expected placeholders.

"Signature template triple" criterion The "signature template triple" criterion check is implemented by using Lemma 2: checking if the value through κ of a template triple tp is in the value through κ of a template graph g is equivalent to checking if the value through κ of the template triple tp is equal to one of the value through κ of one of the template triple of the template graph g. It makes this criterion very similar to check with the "no value loss" criterion. We 1) map the value through κ of each template triple to the list of types that contain a template triple with the same value through κ, 2) list the types that are alone in their list, *i.e.* the types that have a template triple for which its value through κ can not be found for any other type, and 3) check if the list of types that have a signature template triples is the same as the list of types supported by the context.

Note that this algorithm can easily be modified to get the signature template triples by simply modifying the lists contained in $signatureCandidates(aValueThroughKappa)$ from the list of types to a pair composed of the list of types and the list of template triples. If the list of types contains only one element, then all template triples in the list are signature template triples for this type.

Algorithm 13: An optimized way to compute if a context is a PRSC well-behaved context

Input: $rdf \subset RDFTriples$, $ctx \in Ctx^+$
Output: An element of APG or error

Main Function *IsWellBehaved(ctx)*:
  $signatureCandidates \leftarrow \{\}$ `/* A map from image through kappa to types */`
  forall $type \in Dom(ctx)$ **do**
    $uniqueCandidates \leftarrow \{\}$ `/* A map from image through kappa to triples */`
    forall $triple \in ctx(type)$ **do**
      $kappa \leftarrow \kappa(triple)$
      `/* Check violation of the "Element Identification" criterion */`
      if $?self \notin triple$ **then return** *False*
      `/* Other criteria */`
      $signatureCandidates(kappa) \leftarrow signatureCandidates(kappa) \cup \{type\}$
      $uniqueCandidates(kappa) \leftarrow uniqueCandidates(kappa) \cup \{triple\}$
    `/* Check violation of the "No value loss" criterion */`
    $expectedPlaceholders \leftarrow \{(key, valueOf) \mid key \in keys(type)\}$
    if $kind(type) =$ "$edge$" **then**
      $expectedPlaceholders \leftarrow expectedPlaceholders \cup \{?source, ?destination\}$
    forall $(kappa, triples) \in uniqueCandidates$ **do**
      `/* Remove the placeholders of the expectedPlaceholders list that are in unique template triples */`
      if $\exists! triple \in triples$ **then**
        $actualPlaceholders \leftarrow \{term \mid term \in triple \wedge term \in P\}$
        $expectedPlaceholders \leftarrow expectedPlaceholders - actualPlaceholders$
    if $expectedPlaceholders \neq \emptyset$ **then return** *False*
  `/* Check violation of the "Signature template triple" criterion */`
  $signedTypes \leftarrow \{\}$
  forall $(kappa, types) \in signatureCandidates$ **do**
    if $\exists! type \in types$ **then**
      $signedTypes \leftarrow signedTypes \cup \{type\}$
  if $signedTypes \neq Dom(ctx)$ **then return** *False*
  `/* No violation has been detected */`
  return *True*

Complexity analysis

First outer loop The first outer loop loops on the list of types from line 3 to line 17. Its first inner loop loops on the list of template triples of each type in lines 5-9, and performs four operations: 1) computing the value through κ of the template triple in lines 6, 2) checking if *?self* is in the template triple in line 7, 3) adding the type to *signatureCandidates* in line 8, and 4) adding the template triple to *uniqueCandidates* in line 9. All the listed operations have a constant complexity so the complexity of the content of the first inner of the first inner loop is constant and the whole first inner loop has a complexity linear with *BiggestTemplateSize*.

Lines 10 to 12 build the list of expected placeholders for a given type. The *expectedPlaceholders* set contains one placeholder per property key in the current type, and potentially two extra placeholders for the source and the destination. Its size is linear with *TypeComplexity* and so is the complexity of building it.

The second nested loop from line 13 to line 16, aims to remove the placeholders from *expectedPlaceholders* as they are spotted in unique template triples, the size of *uniqueCandidates* is in the worst case equal to *BiggestTemplateSize*. This case occurs when all template triples are *unique*. In the removal nested loop, listing the list of placeholders in the template triple is considered constant, and the complexity of removing elements from *expectedPlaceholders* depends on the size of *actualPlaceholders*. Because the size of the triples is bounded by a constant, so is the size of *actualPlaceholders*, and the time to remove all its elements from *expectedPlaceholders* at each step of the loop. Therefore, the complexity of the whole loop from line 13 to line 16 is linear with the number of triples *BiggestTemplateSize*.

To summarize, the cost of 1) the first nested loop is linear with the number of template triples in the type, *i.e.* *BiggestTemplateSize*, 2) computing *expectedPlaceholders* is *TypeComplexity*, and 3) the expected placeholder removal is *BiggestTemplateSize*. So the overall cost of the first outer loop is $\mathcal{O}(NbTypes * (BiggestTemplateSize + TypeComplexity))$.

Rest of the algorithm The *signatureCandidates* set has at most *NbTypes* $*$ *BiggestTemplateSize*: it is the case where all template triples in the context are signature template triples. All operations of the second outer loop are in constant time, so the complexity of this loop is $\mathcal{O}(NbTypes * BiggestTemplateSize)$

The *signedTypes* set has at most *NbTypes*, so the complexity of checking if *signedTypes* is equal to $Dom(ctx)$ is in the worst case scenario linear with *NbTypes*.

Final complexity The added cost of the other instructions is overshadowed by the complexity of the first outer loop, so the final complexity of the optimized *IsWellBehaved* function presented in Algorithm 13 is:

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

For reference, the naive cost presented in Section 5.5.3 was:

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

We see that we saved a *NbTypes* factor and a *BiggestTemplateSize* factor, respectively, on the two terms of the complexity. This is achieved by using efficient data structures for checking the "signature template triple" and "no value loss" criteria. Note that these data structures have a size linear to the complexity indicators, therefore comparable to that of the context itself. The improvement in time complexity comes with no additional cost in space complexity.

Finding the signatures As discussed previously, the same algorithm can be used to build the list of the signature template triple of each type. This can be done with no added complexity as each *signatureCandidates* map is simply filled with one extra information: a template triple that has the corresponding value through κ.

Note that the *TypeComplexity* factor comes from checking the "No value loss" criterion. If one is only interested in implementing a function that looks for one signature template triple for each type, all instructions related to the "no value loss" criterion may be removed, lowering the complexity of the function to $\mathcal{O}(NbTypes * BiggestTemplateSize)$.

5.6.2 Associating the elements of the future PG with their types

Algorithm 10 is inefficient in its usage of signature template triples. Instead of "discovering" the signature template triple of each type for each explored data triple, Algorithm 14 maps the signatures to their types and then uses the map to find if the data triple corresponds to a signature or not.

Algorithm 14: Associate the elements of the future PG with their types (optimized version of Algorithm 10)

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

```
1  Function FindTypeOfElements(rdf, ctx, Elements):
       /* Build the map from signatures to their type                    */
2      signed ← {κ(sign_ctx(type)) ↦ type | type ∈ Dom(ctx)}
       /* Explore the RDF graph                                          */
3      candtypes_nodes ← {}
4      candtypes_edges ← {}
5      forall triple t ∈ rdf do
6          kappa ← κ(t)
7          if kappa ∉ Dom(signed) then
8              Continue
9          type ← signed(kappa)
10         foreach m ∈ t ∧ m ∈ B do
11             if kind(type) = "node" then
12                 candtypes_nodes(m) ← candtypes_nodes(m) ∪ {type}
13             else
14                 candtypes_edges(m) ← candtypes_edges(m) ∪ {type}
       /* Decide the type of each PG element                             */
15     typeof ← {}
16     forall element m ∈ Elements do
17         if (∃!type ∈ candtypes_nodes(m)) or (∃!type ∈ candtypes_edges(m) and
             candtypes_nodes(m) = ∅) then
18             typeof(m) ← type
19         else
20             raise Error(No type found)
21     return typeof
```

Differences with the original version

The main difference between the two algorithms 10 and 14 is that the exploration of all triples in the RDF graph is mutualized for all PG elements/blank nodes: the RDF graph is explored only once.

The *candtypes* sets are computed for all PG elements at the same time, instead of by pair of two $candtypes_{nodes}/candtypes_{edges}$ for a given PG element like in the original algorithm.

The optimized algorithm fully relies on the fact that a signature template triple is a template triple for which its value through κ is not shared by any other type by definition. It is reflected by the usage of the map from signatures to types to directly find if the data triple may help to find the type of the element or not.

Complexity analysis

Like the unoptimized version, we assume that calls to the $sign_{ctx}$ are constant because we already checked if the context is a PRSC well-behaved context. If it is not the case, the complexity discussed in Section 5.6.1 should be added to the complexity calculated in this section.

Building the map from the value through κ of the signatures to the corresponding type has a complexity linear with the number of types *NbTypes*.

In the exploration of the RDF graph loop, all operations are constant. Even the loop that consists in looking for all blank nodes in a given triple is done in constant time because we consider that a triple has a bound depth. So the cost of exploring the RDF graph is linear with the size of the RDF graph *NbTriples*.

The operations performed in the loop to decide the type of each PG element are all done in constant time. The loop iterates on each PG element, so the cost of the complete loop is linear with *NbOfPGElements*.

The final complexity of this optimized version of the *FindTypeOfElements* function is:

$$\mathcal{O}(NbTypes + NbTriples + NbOfPGElements)$$

The previous version has an $\mathcal{O}(NbOfPGElements * NbTriples * NbTypes)$ complexity. The gain mostly comes from un-nesting the different loops.

5.6.3 Producing the PG

Algorithm 12 is inefficient in how it looks for the values corresponding to the placeholders. Each call to the *extract* function searches in the entire sub-graph corresponding to the element. Each call to *extract* also has to discover which template triples are *unique*. Algorithm 15 addresses these two issues.

Algorithm 15: Produce a PG from the previous analysis of the elements and triples (optimized version of Algorithm 12).

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

Output: A member of APG or error

Function $buildpg(ctx, Elements, typeof, builtfrom)$**:**
- g is initialized to the empty PG
- **forall** $b \in Elements$ **do**
  - `/* Compute the unique template triples */`
  - $tps \leftarrow ctx(typeof(b))$
  - $uniques \leftarrow \{\kappa(tp) \mapsto tp \mid unique(tp, tps)\}$
  - `/* Explore the RDF graph */`
  - **forall** $td \in builtfrom(b)$ **do**
    - **if** $\kappa(td) \in Dom(uniques)$ **then**
      - $merge(g, b, td, uniques(\kappa(td)))$
  - `/* Ensure that all placeholders are filled */`
  - **forall** $key \in keys(typeof(b))$ **do**
    - **if** $(b, key) \notin Dom(properties_g)$ **then**
      - **raise** *Error(Missing a property key)*
  - **if** $kind(typeof(b)) =$ "*edge*" **then**
    - **if** $b \notin Dom(src_g) \vee b \notin Dom(dest_g)$ **then**
      - **raise** *Error(Missing source or destination)*
  - `/* Leftovers */`
  - $labels_g(b) \leftarrow labels(typeof(b))$
  - **if** $kind(typeof(b)) =$ "*edge*" **then**
    - $N_g \leftarrow N_g \cup \{src_g(b), dest_g(b)\}$
    - $E_g \leftarrow E_g \cup \{b\}$
  - **else**
    - $N_g \leftarrow N_g \cup \{b\}$
- **return** g

Algorithm 16: Utility functions of Algorithm 15

```
   /* Merges in the PG g the data that can be extracted from the triple td considering
      it was produced from element b and the template triple tp                        */
 1 Function merge(g, b, td, tp):
 2     extractedPairs ← extractPairs(td, tp)
 3     forall (placeholder, term) ∈ extractedPairs do
 4         if placeholder = ?source then
 5             if b ∈ Dom(src_g) ∧ src_g(b) ≠ term then
 6                 raise Error(Inconsistent value for ?source)
 7             src_g(b) ← term
 8         else if placeholder = ?destination then
 9             if b ∈ Dom(dest_g) ∧ dest_g(b) ≠ term then
10                 raise Error(Inconsistent value for ?destination)
11             dest_g(b) ← term
12         else
13             (key, _) ← placeholder
14             if (b, key) ∈ Dom(properties_g) ∧ properties_g(b, key) ≠ toLiteral^-1(term) then
15                 raise Error(Inconsistent value for ?destination)
16             properties_g(b, key) ← toLiteral^-1(term)
   /* Extract the list of terms produced from the placeholders.  Recursive function on
      triples/terms.                                                                   */
17 Function extractPairs(tp, td):
18     if td ∈ RdfTriples ∧ tp ∈ Templates then
19         (ts, tp, to) ← tp
20         (s, p, o) ← td
21         return extractedPairs(ts, s) ∪ extractedPairs(tp, p) ∪ extractedPairs(to, o)
22     else if td ∈ L then
23         if tp ∈ L then return ∅
24         else if tp ∈ PL then return {(tp, td)}
25     else if td ∈ B then
26         if tp = ?source then return {(tp, td)}
27         else if tp = ?destination then return {(tp, td)}
28         else if tp = ?self then return ∅
29     else if td = tp then return ∅
30
31     raise Error(tp did not generate td)
```

Explanation of the new algorithm

The *extractPairs* is in charge of extracting the different information contained in a data triple assuming it has been produced by the given template triple. It is a recursive function that can be seen as the inverse of the β function. Note that it can return several times the same pairs, or may even in principle return multiple pairs with the same first member but a different second member. For example, consider the template triple $(?source, rdf{:}type, ?source)$ and the data triple $(\_{:}a, rdf{:}type, \_{:}b)$. In this case, the set $(?source, \_{:}a), (?source, \_{:}b)$ will be returned.

However, as we supposed that the RDF graph is produced from the *prsc* function, and in particular by using a PRSC well-behaved context, this case will never occur.

The *merge* function is in charge of extracting the different information contained in the data triple and filling the PG with it. Using the result of *extractPairs*, it merges the data in the right places (the src_g, $dest_g$ and $properties_g$ functions of the PG being constructed), ensuring that the inserted data remains consistent. It is where the erroneous example presented above is caught.

The new *buildpg* function, for each PG element, 1) computes the *unique* template triples, 2) for each data triple that has the same value through κ as a *unique* template triple calls the *merge* function, 3) ensures that all missing piece of information has been filled, and 4) adds the remaining data.

Complexity analysis

Like the β and the κ functions, the *extractPairs* function has a constant complexity as it is a recursive function on triples and atomic terms.

As the *merge* function loops on the output of the *extractPairs* function, and only performs constant operations, it is also assumed to be run in constant time. This is because each pair of template triple and data triple with a bounded nesting depth will only contain a bounded amount of elements to merge in the PG.

Let us now study the new *buildpg* function from the point of view of one given element, *i.e.* inside the outermost loop:

- The first step lines 4-5 consists in building the list of template triples that are *unique*. Finding the template graph, *i.e.* calling the *ctx* function, has an $\mathcal{O}(TypeComplexity * \ln(TypeComplexity))$ complexity as discussed in Section 5.4.5. The computation of the *uniques* set, in line 5, can be performed using the same method as Algorithm 12, with a complexity of $\mathcal{O}(BiggestTemplateSize)$ by reusing the ideas presented in the optimized PRSC Well-behaved context check in Algorithm 13 of Section 5.6.1. The final complexity of this step is $\mathcal{O}(TypeComplexity * \ln(TypeComplexity) + BiggestTemplateSize)$.
- The second step lines 6-8 consists in extracting data from exploring the triple from the graph. As calling *merge* has a constant complexity, this step has an $\mathcal{O}(NbTriples)$ complexity. Note that in the case where the RDF graph has been produced by the PRSC well-behaved context ctx, $builtfrom(b)$ will return at most $BiggestTemplateSize$ triples, so the complexity drops to $\mathcal{O}(BiggestTemplateSize)$ complexity.
- The third step lines 9-14 ensures that all required data is in the PG, which has an $\mathcal{O}(TypeComplexity)$ complexity.
- The fourth step has a constant complexity.

As the *buildpg* function loops on all PG elements, its complexity is

$$\mathcal{O}(NbOfPGElements*(TypeComplexity*\ln(TypeComplexity)+BiggestTemplateSize+NbTriples))$$

The complexity of the previous non-optimized version of *buildpg* is

$$\mathcal{O}(NbOfPGElements*TypeComplexity*(\ln(TypeComplexity)+BiggestTemplateSize^2*NbTriples))$$

By un-nesting the different loops and by using appropriate data structures, we are able to transform some factor into additions.

5.6.4 Complexity of the optimized RDF to PG function

By using the new versions of the *FindTypeOfElements* and the *buildpg* functions, the new complexity of the *RDFToPG* function is:

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

The complexity of the non optimized version was:

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

The complexity of the optimized version is way more acceptable. Most of the cost comes from the final *buildpg* function. The biggest factor in practice is the $NbOfPGElements * NbTriples$ part, which in the case of RDF graphs generated from PRSC well-behaved context will be further lowered as the number of triples actually processed for each element will be bound by *BiggestTemplateSize* instead of *NbTriples*. It means that the reversion algorithm is very efficient and can scale with large graphs.

5.7 Extensions

In this section, multiple extensions are proposed to make the *prsc* function more user-friendly.

5.7.1 Edge-unique extension

In many cases, there is only one edge of a certain type 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.

Table 5.11: A context for the Tintin PG with the since property

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

Consider the Tintin PG exposed in Figure 5.1 and the context exposed in Table 5.11, which uses RDF-star to convert the "since" property. The output of PRSC from those two inputs is

exposed in Listing 5.4. 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.

Listing 5.4: The output of PRSC for the Tintin PG and the context exposed in Table 5.11

```
% 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 .
```

Definition 41 [Edge-unique extension]

a) In a context ctx, an **edge-unique type** $edgeunq$ is an edge type such that:

- $ctx(edgeunq)$ complies with the *no value loss* criterion 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:

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

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 36 or b) is an edge-unique type.

- For every two BPGs, $pg1$ and $pg2$, such that $prscEdgeUnique(pg1, ctx) = prscEdgeUnique(pg2, ctx)$, $pg1$ and $pg2$ are isomorphic.
- It is possible to define an algorithm such that $\forall pg \in BAPG$, from the RDF graph $prscEdgeUnique(pg, ctx)$, the algorithm computes a PG pg' such that $prscEdgeUnique(pg, ctx) = prscEdgeUnique(pg', ctx)$, *i.e.* from the produced RDF graph and the context, it is possible to compute a PG that is isomorphic to the original one.

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

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

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

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

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

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.

□

Remark 23 [Reversibility of the union of two contexts]
We proved for Theorem 7 that given a context ctx that can be split into two contexts ctx_w and ctx_u where ctx_w is a PRSC well-behaved context and ctx_u is the "edge-unique" part of the context, the context $ctx = ctx_w \cup ctx_u$ is reversible.

However, the proof is based on the fact that:

- The image through κ of all template triples in the context ctx_w and the image through κ of all template triples in the context ctx_u are disjoint.
- The contexts ctx_w and ctx_u are reversible.

Let us now consider the case of two contexts ctx_a and ctx_b such that the images through κ of their template triples are disjoint and such that they are reversible, possibly up to an isomorphism.

If the two contexts are compatible (no type is defined in both contexts), the function $ctx_a \cup ctx_b$ is a context and is reversible, up to an isomorphism if ctx_a or ctx_b is reversible up to an isomorphism.

By consequence, similarly to how we showed that contexts that use the edge-unique extension are still reversible, the range of reversible contexts can be increased by splitting the context into multiple parts and proving that each one is reversible.

5.7.2 Default context

The definition of a PRSC context forces the user to map each type present in the PG. By consequence, it is impossible to write in practice a PRSC context that works for all PGs as it would require to write an infinite number of rules. In this section, we introduce the notion of a default context.

Let us suppose that we have an injective function $toiri : Str \mapsto I$ that maps all possible labels to distinct IRIs.

Definition 42 [Default context]
The context $ctx_{default}$ is defined for all types as follows:

$$ctx_{default} : type \mapsto \left\{ \begin{array}{ll} \text{if } kind(type) = \text{“}node\text{”}, & (?self, rdf{:}type, pgo{:}Node) \\ \text{if } kind(type) = \text{“}edge\text{”}, & (?self, rdf{:}type, pgo{:}Edge) \\ \forall label \in labels(type), & (?self, {:}label, toiri(label)) \\ \forall key \in keys(type), & (?self, toiri(key), (key^{valueOf})) \\ \text{if } kind(type) = \text{“}edge\text{”}, & (?self, pgo{:}start, ?source) \\ \text{if } kind(type) = \text{“}edge\text{”}, & (?self, pgo{:}end, ?destination) \end{array} \right\}$$

$ctx_{default}$ can be used for all PGs, *i.e.* $\forall pg \in APG, ctx_{default} \in Ctx_{pg}$.

Lemma 6
The default context is not a PRSC well-behaved context, *i.e.* $ctx_{default} \notin Ctx^+$

Proof. Consider back the running example (Figure 5.1, and especially 1) the node for Tintin that has a Person label and two properties: the name property and the job property; and 2) the node for Snowy that only has a name property.

The type corresponding to Tintin has four template triples: one to add the type $pgo{:}Node$, one for the label and two for the properties, in particular the template triple $(?self, {:}label, \text{“}name\text{”}^{valueOf})$ for the name property. The type corresponding to Snowy has two templates triples, one to add the type $pgo{:}Node$, and the template triple $(?self, {:}label, \text{“}name\text{”}^{valueOf})$ for the name property.

As the latter template graph is a subset of the former template graph, the default context is not a well-behaved PRSC context. □

Lemma 7 [The default context is reversible]

Proof. The default context trivially satisfies the *no value loss* and the *element provenance* criteria from well-behaved contexts. The only part of the reversion algorithm (Algorithm 9 that needs to be changed to be used for the default context is the type identification method (the *FindTypeOfElements* function from Algorithm 10).

However, it is possible to write a new function as the retrieval of the type is trivial: with pg the original PG and rdf the built RDF graph, for any element m, its type $typeof_{pg}(m)$ is

- $$kind(typeof_{pg}(m)) = \begin{cases} \text{“}edge\text{”} & \text{if } \exists n \in B, (m, pgo{:}start, n) \\ \text{“}node\text{”} & \text{otherwise} \end{cases}$$
- $labels(typeof_{pg}(m)) = \left\{ toiri^{-1}(iri) \mid iri, (m, :label, iri) \in rdf \right\}$
- $keys(typeof_{pg}(m)) = \left\{ toIri^{-1}(iri) \mid iri, \exists value \in L, (m, iri, value) \in rdf \right\}$

□

It is possible to apply the idea introduced by Remark 23 of combining two contexts, a user-defined context and the default context, to produce a context that works for any PG in the case where we want to use PRSC but still want to be able to convert any arbitrary PG. Formally, the new context would be defined as the union of 1) the user-defined context and 2) the default context deprived of the types defined by the user-defined context, *i.e.* a version of the default context in which the types in the user-defined context are removed from the domain. From the same Remark 23, as the default context is reversible (Lemma 7), if image through κ of all templates triples in the user-defined context are disjoints from the ones of the default context, and if the user-defined context is reversible, then the context composed of both contexts is still reversible.

5.7.3 IRI Property Graphs

In Section 3.6.2, we discussed using Property Graphs with Blank Nodes to build RDF graphs. The process to convert PGs into RDF graphs was, 1) build an isomorphic PG for which each element has been mapped to a blank node, and 2) then each element is converted depending on its type.

However, a blank node has no tangible identity, and using blank nodes reduces the utility of the produced RDF graph in the context of Linked Data, as its elements can not be linked from other graphs.

Very few IRIs are used when converting a PG into an RDF graph: the used IRIs are limited to the ones that are present in the PRSC context. Instead of using blank nodes for the PG elements, we could use any IRI that is not used in the context. These IRIs can be forged using the values of some properties.

Definition 43 [Used IRIs]
The *usedIris* function extracts all the IRIs used in a template graph:
$usedIris : Templates \to 2^I$
$usedIris(tps) = \{i \mid i \in I \wedge \exists tp \in tps, i \in tp\}$
The *usedIris* function is extended to contexts to provide the list of IRIs used in the

context:

$$\forall ctx \in Ctx, usedIris(ctx) = \bigcup_{tps \in Img(ctx)} usedIris(tps)$$

When reversing an RDF graph produced by *prsc* using a context ctx, the only IRIs that can be generated from the template graphs themselves are the ones explicitly written in the context, *i.e.* $FI_{ctx} = usedIris(ctx)$. Let us call theses IRIs "fixed iris".

By replacing in the proofs presented in Section 5.5 I with FI_{ctx}, the terms in the set of *variable IRIs* $VI_{ctx} = B \cup (I - FI_{ctx})$ may never be produced from the non placeholders terms of the template graphs. These variables IRIs can play the role of the set B, *i.e.* we may substitute all mentions of B with VI_{ctx}.

Then, by extending the notion of BPGs to also allow IRIs as the PG elements identity, we are able to produce RDF graphs that contain IRIs as PG elements identifiers, and for any BPGs that do not contain a PG element that is part of the set FI_{ctx}, *i.e.* they are all part of the set VI_{ctx}, if ctx is a PRSC well-behaved context, then the conversion is reversible.

Example 28
In our running example about the PG about Tintin, the PG TT, consider that we want to build IRIs for all nodes in the form `ex:individual/{name}` where `{name}` is substituted with the value of the property name. For example, Tintin is now mapped to the IRI `ex:individual/Tintin` instead of an arbitrary blank node.

Using any PRSC well-behaved context such as the one presented in Table 5.8 that do not map any IRI to the `ex:individual/` namespace will still produce a reversible RDF graph: during the reversion, instead of only considering the elements in the set B, elements in the `ex:individual/` namespace may also be considered as PG elements.

The main use case would be to generate IRIs by using a prefix in a given namespace and a suffix that depends on a property value, like Example 28 above. It could also be imagined that multiple properties can be used to generate an IRI, for example to use both the given name and the family name of the persons in the PG to generate an IRI to identify the persons. However, generating IRIs from property values comes with an additional constraint: in addition to not collide with the IRIs in the context, the IRI generated from the property values must be different for all PG elements. In other words, the tuple composed of all properties used in an IRI must be a primary key (in its database definition) in the PG.

5.8 Conclusion

In this chapter, we presented PRSC, a converter from PG to RDF graphs based on a PRSC context. The PRSC context is written by the user and maps the different types in the PG to template graphs. Then the PRSC algorithm transforms the PG into an RDF graph by looking at each PG element and by using for each of them the appropriate template graph depending on the type of the PG element. We defined a subclass of contexts named PRSC well-behaved contexts for which we formally proved the reversibility, and provided the related algorithm. We showed that the reversion algorithm can be optimized to a version with a complexity that makes it usable with large graphs. We then introduced some extensions to this work, in particular we extended the notion of PRSC well-behaved contexts to consider PGs for which certain edge labels may be edge-unique.

The work on PRSC would be extended further to improve its usability, in particular on properties:

- The system currently does not allow optional properties. Two types that differ only by the fact that the second type has an extra property are considered to be distinct types. This is especially annoying in the case of PRSC well-behaved contexts where the "signature template triple" criterion forces the user to have a distinct triple in the template graphs of these two types. To tackle this issue, the types could be recognized not by using a single signature template triple, but through an approach based on FCA (formal concept analysis), or using existing tools like ShEx to detect type overlapping and/or recognize the types.
- Meta properties are currently not supported. Supporting them does not add any challenge as meta properties can be considered to be a fancy method to name properties. For example the property named "name.since" can be considered to be the meta property "since" on the property "name".
- Multi valued properties, *i.e.* two properties with the same key, is also not supported. Again, supporting this feature introduces no extra challenge to produce the RDF graph as the same template triple could be used for the different values of the properties that share the same property key. However, note that for the reversion, a special attention must be paid to multi-valued properties with both the same key and value to avoid collapsing them in the same triple, for example producing two $(\_:(tintin), foaf{:}name, \text{"}Tintin\text{"})$ triples that are actually collapsed into one triple. Note that this issue is very similar to the issue of *edge-unique types* where the property holder is similar to the edge source, and the property value is similar to the edge destination.
- However, supporting both meta-properties and multi- valued properties adds new challenges for the RDF to PG reversion algorithm. Indeed, consider a node that has two "name" properties and a meta property on both of them. When reverting the RDF graph to a PG, the system must be able to identify to which "name" property each "since" meta property was attached to.

Compared to PREC-C, PRSC faces some limitations:

- As there are no placeholders corresponding to the RDF node of PG properties, *i.e.* the elements of the set P in the GPG formalism, it is impossible to produce an RDF graph with a unique identifier for properties. This is especially a problem when trying to reproduce some ontologies, for example the PGO ontology that uses a blank node for each property as the value of `pgo:hasNodeProperty` and `pgo:hasEdgeProperty` as seen in Figure 4.3.
  Being able to use the RDF node of PG properties is a solution to be able to support the reversion of a conversion that supports both meta and multi properties.
- The list of supported types must be specified by the user, which may be tedious for PGs with heterogeneous types in the PG. The lack of any default mechanism blocks the user from producing a PG without defining a complete context that supports everything in the PG.

Over PREC-C, PRSC also has many advantages:

- The ontology to describe PRSC contexts is simpler than the ontology to describe PREC-C: it contains fewer terms.
- Moreover, the context is self-contained: it does not have any implicit rules like the default template graph of PREC-C.
- As for a given context, some PGs may not be valid, PRSC includes some form of schema validation which is less error-prone. With PREC-C, users may produce unexpected triples

from a PG that contain data that they are not aware of without the converter explicitly noticing them about these data.

Chapter 6

Shacled Turtle: a general purpose autocompletion engine

Chapters 3, 4 and 5 introduced PREC, a PG to RDF converter that relies on a context, a file written in Turtle-star that describes how to convert the different PG entities into RDF triples.

The two rule-sets used by PREC, PREC-C and PRSC, rely on rules that describe the target and the production. However, users may not be fluent in writing a context.

When users are facing a new ontology, they are most of the time expected to extensively read its documentation. Moreover, ontologies are often described in RDF themselves using ontology languages like RDFS or OWL. Validation schema languages like SHACL and ShEx have been proposed to represent constraints and verify that some graphs have some properties, for example that all instances of a class have a given property. It should be noted that these two kinds of schemas essentially describe the links between the different objects used in an RDF graph, and could be used for this property.

Like other ontologies, the PREC vocabulary is also described by a user-friendly page and by an RDF document suitable for machine processing. To help users use the PREC ontology, we propose to use the RDF document for computers to power up an auto-completion engine.

Listing 6.1: A subgraph of the PREC shape graph

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix prec: <http://bruy.at/prec#>.
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

# PRSC ruleset
prec:PRSCNodeRule a rdfs:Class, sh:NodeShape ;
  sh:property [ sh:path prec:label ] ;
  sh:property [ sh:path prec:propertyKey ] ;
  sh:property [ sh:path prec:produces ] .

prec:PRSCEdgeRule a rdfs:Class, sh:NodeShape ;
  sh:property [ sh:path prec:label ] ;
  sh:property [ sh:path prec:propertyKey ] ;
  sh:property [ sh:path prec:produces ] .


# PREC-C ruleset
prec:PropertyRule a rdfs:Class, sh:NodeShape ;
  sh:property [ sh:path prec:propertyKey ] ;
  sh:property [ sh:path prec:propertyIRI ] ;
  sh:property [ sh:path prec:label ] ;
  # ...
.
```

Listing 6.1 shows a sub-graph of the SHACL shape graph describing the PREC ontology.

As we can see, the predicates related to the different rule types are different, and this document could be used to provide auto-completion.

Instead of building an auto-completion tool that would only work for PREC, this chapter focuses on building a general purpose auto-completion tool using schemas, both RDFS as an inferential schema and SHACL as a validation schema, then evaluates the tool for other existing ontologies and finally concludes with a refocus on PREC.

The work on this chapter has been published at the VOILA! 2022 workshop [78]. The main difference between this chapter and the article is a re-contextualization of the work with PREC. In particular, this chapter uses writing PRSC as a running example and offers a final discussion on the usability of Shacled Turtle for PREC.

6.1 Shacled Turtle usage example

Shacled Turtle is implemented as a Code Mirror 6[1] extension that provides support for the Turtle language.

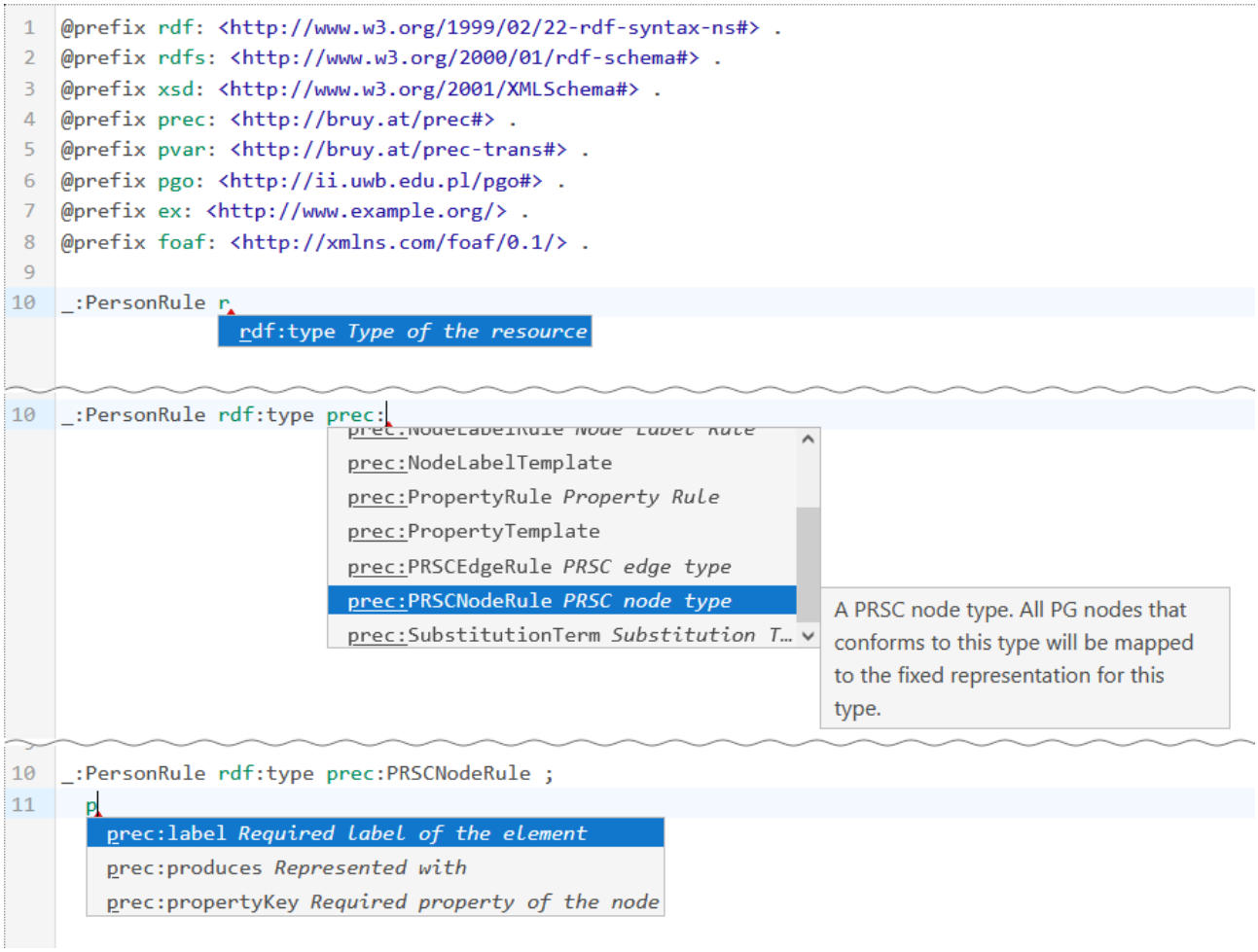


Figure 6.1: Shacled Turtle helps writing a context by exposing the types and then the different predicates related to the chosen type and its documentation.

[1] https://codemirror.net/6/

The selling key-point of Shacled Turtle is the auto-completion module. While most advanced editors suggest all terms from the ontology, Shacled Turtle narrows the list to the parts of the ontology that are related to the currently edited resource.

We consider that most resources in an RDF graph must be typed. When the type of a resource is known, it is likely that the predicates related to the known types will be used to describe it.

Figure 6.1 shows a concrete usage example: we first write a resource named `_:PersonRule`. As it has not yet any type, Shacled Turtle suggests the term `rdf:type` and then the list of all types that exist in the PREC ontology. From this list, the user chooses `prec:PRSCNodeRule`. Then, they write a new triple, and Shacled Turtle is now able to list the predicates that are related to node rules in the PRSC ruleset: `prec:label`, `prec:produces` and `prec:propertyKey`. The engine does not show any of the predicates that exist in the PREC ontology but are unrelated to `prec:PRSCNodeRule`, like `prec:labelIRI` or `prec:propertyIRI`.

6.2 Shacled Turtle architecture

Figure 6.2 shows the general architecture of Shacled Turtle.

0. Before the interaction starts, a preprocessing phase is performed. The content of a *schema graph* is converted into *inference rules* and *suggestion rules* by the *schema to rules converter*. This schema graph can be written either in RDFS, in SHACL or a mix of both.

1. When the user is writing an RDF graph, the *inference engine* uses all *complete triples* to deduce the types of all resources and the list of shapes that these resources should comply with. These results are stored in the *meta graph*.

2. When the user is writing a new *incomplete triple*, after the subject has been written, *i.e.* on writing the predicate or on writing the object, the *suggestion engine* queries the *meta graph* for the list of all the types and shapes of the subject. It will then return to the user:

    - If the incomplete triple only has a subject, the list of all predicates related to the types and shapes of the subject.
    - If the incomplete triple has a subject and a predicate, a list of resources depending on the types and shapes of the subject and the predicate[2].

In Section 6.3, we describe the basics of all the components used by the *interaction loop*. In particular, we specify how the rule systems used by the *inference engine* and the *suggestion engine* work. In Section 6.4, we describe how the *preprocessing* translates the schema graph into *inference and suggestions rules*.

6.3 The interaction loop

The **interaction loop** comprises all operations performed when the user uses the editor.

[2]Note that for some predicates like `rdf:type`, the suggestion may be independent of the list of types and shapes of the subject.

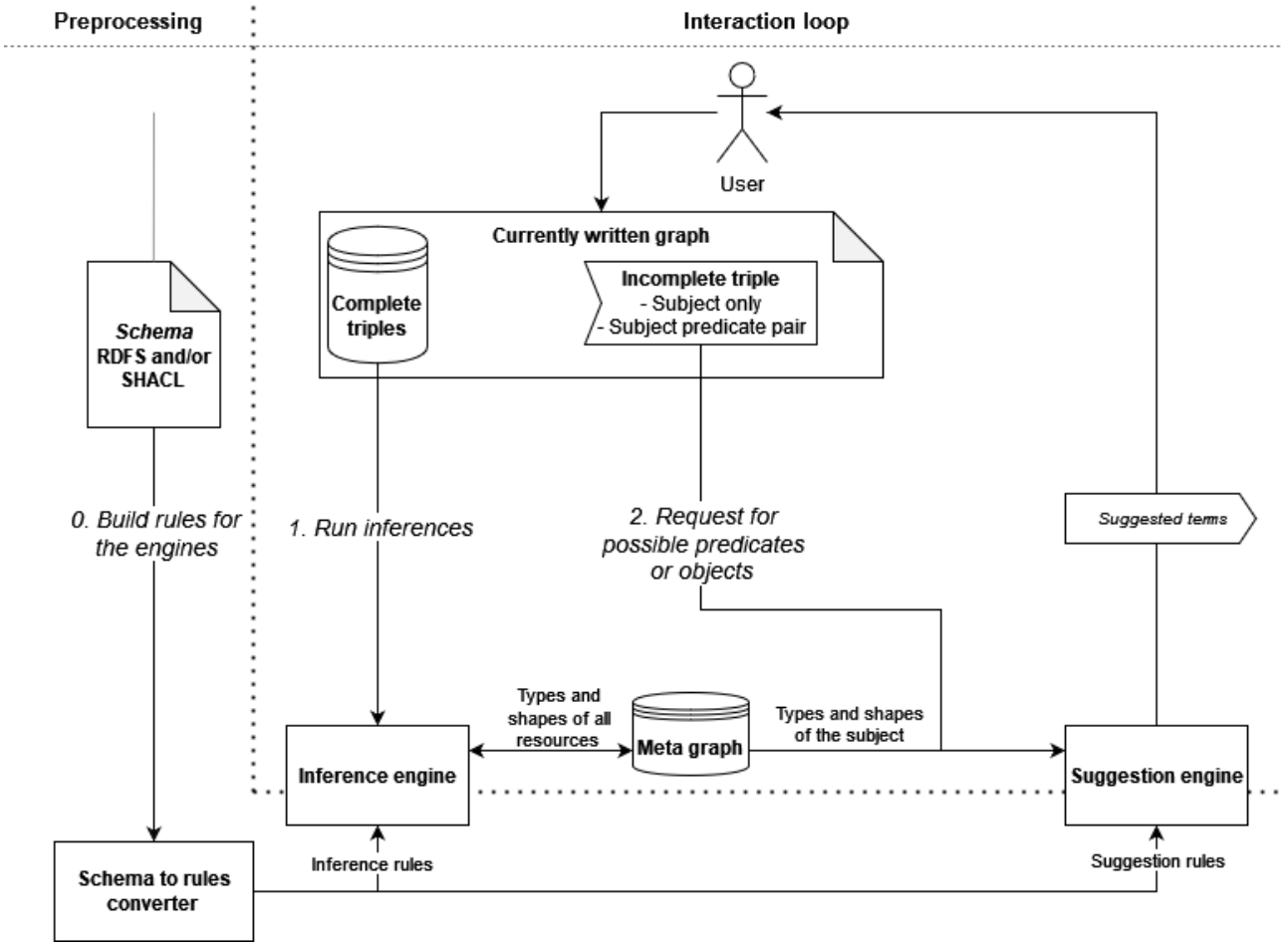


Figure 6.2: The different components of Shacled Turtle

6.3.1 The graphs

During the interaction loops, two different graphs are used:

- The *currently written graph* is the graph in the text editor. It is composed of two different parts:
  - The *completed triples*, triples for which the subject, the predicate and the object are known. These triples are used to power up the *inference engine* and produce triples for the *meta graph*.
  - The *incomplete triple* that the user is currently editing. If the subject of this incomplete triple is known, the *suggestion engine* and the *meta graph* will be requested to provide a list of *suggested terms*. If some other triples are incomplete, they are ignored by the engine.
- The *meta graph* is the graph that stores triples produced by the *inference engine*. Its role is to store, for each resource, the list of all known types, and the list of shapes that the resource should comply with. The content will then be used by the *suggestion engine*, in particular to know the list of types and shapes of the subject of the *incomplete triple*.

6.3.2 The inference engine

The purpose of this engine is to deduce types using an RDFS ontology and to list the SHACL shapes each resource should comply with.

These inferences are specified by *inference rules* (see Tables 6.1 and 6.3). These rules go beyond the ones defining RDFS semantics, but do not need to capture the full semantics of SHACL as we are not aiming at validating the graph.

Each inference rule has the form:

$$\frac{DataTriple? \quad SourceMetaTriple?}{ProducedMetaTriple}$$

where:

- The body is composed of
  - $DataTriple?$: 0 or 1 complete triple from the *currently written graph*.
  - $SourceMetaTriple?$: 0 or 1 triple from the *meta graph*.
- The head is a triple called $ProducedMetaTriple$ that will be stored in the *meta graph*, and its predicate must either be `rdf:type` or `:pathsOf`[3].

Because inference rules can only produce triples with `rdf:type` or `:pathsOf` as the predicate, only such triples are stored in the meta graph.

6.3.3 The suggestion engine

In the same way, the system relies on a set of rules to deduce the possible suggestions at run-time, from the meta graph and the incomplete triple.

We suppose that $?s, ?p, ?o$ are variables, terms that depend on the *currently written graph* *i.e.* when the rule is used, and A and P are IRIs chosen when the rule is built.

Each suggestion rule has the form:

$$\frac{IncompleteTriple \quad MetaTripleCondition?}{Suggestion}$$

where:

- The $IncompleteTriple$, extracted from the *currently written graph*, is either
  - $(?s, \dots, \dots)$ for applying the rule when only the subject of the *incomplete triple* is known.
  - $(?s, A, \dots)$ for applying the rule when both the subject and predicate are known.
- The $MetaTripleCondition?$ is extracted from the *meta graph* and is either:
  - A triple pattern of the form $(?s, P, ?o)$ that is searched in the *meta graph*, where $?s$ is the subject occurring in $IncompleteTriple$. The value of P is fixed for each rule and is either equal to `rdf:type` or `:pathsOf`.
  - "No info on $?s$", for applying the rule only if there are no type or shape known for the resource `?s`.

[3]The triple $(u, :pathsOf, s)$ means that for the resource u we should suggest the paths specified by the shape s.

  - *none* when no condition holds on the meta graph content.

- The *Suggestion* is either
  - $suggest(A)$ to add A to the list of suggested terms.
  - $suggestAll(?p, ?o)$ to add to the list of suggested terms all resources α such that $(\alpha, ?p, ?o)$ is in the *meta graph*.

6.4 The preprocessing

We now describe the *preprocessing*, which is the step where the *schema to rules converter* converts the *schema* graph into *inference* and *suggestion rules* for the eponymous engines. It uses two kinds of transformations: rules that are built by searching all triples with a certain pattern in the schema graph, and SHACL paths whose recursive nature is handled by using finite state automata.

6.4.1 Rules built by looking up some triple patterns

Table 6.1 exposes the list of inference and suggestion rules that are generated from the schema graph. It can be seen as a list of *meta rules*, *i.e.* rules that tell how to generate the *inference and suggestion rules*:

- The first row exposes the axiomatic inference and suggestion rules. These axioms include some of the RDFS rules, for example the second inference rule is about deducing that a resource used as a type has the `rdfs:Class` type. In terms of suggestions, the generated rules are relative to the types, *i.e.* suggesting `rdf:type` as a predicate and suggesting types when the predicate is `rdf:type`
- The next two rows are standard RDFS rules about deducing the type of a resource used either as the subject or the predicate. In the `rdfs:domain` row, it should be noted that suggestion rules are generated to suggest predicates whose domain is known when the subject of an incomplete triple has no known types or shapes, as choosing this predicate will allow the *inference engine* to deduce a type for the subject.
- On the fourth and fifth row, corresponding to the second part of the table, the *suggestion engine* generate suggestion rules for `schema:domainIncludes` and `schema:rangeIncludes`. Unlike `rdfs:domain` and `rdfs:range`, these predicates can not be used for inference as they do not enforce any type relation.
- On the third part of the table, rules related to shape targets are generated. The generated inference rules are used to compute to which shape each resource that appears in the *currently written graph* are supposed to comply, which will serve us to suggest the predicates related to these shapes for these resources. Suggestions rules are also generated for the shape targets, based on the observation that the pair `rdfs:domain` and `rdfs:range`; and the pair `sh:targetSubjectsOf` and `sh:targetObjectsOf` play a similar role in their respective schema vocabulary.

Note that the purpose of this tool is neither to infer all possible suggestions, nor to validate the graph, but to make suggestions that are as relevant as possible. This is a subjective criterion, as having either too few or too many suggestions would make the tool less useful. We will discuss this further in Section 6.7.

Table 6.1: Transformation of triples in the schema graph into inference and suggestion rules.

Triple in schema graph	***Inference rules***	***Suggestion rules***
	$\frac{(?u, rdf{:}type, ?t) \quad none}{(?u, rdf{:}type, ?t)}$ $\frac{none \quad (?u, rdf{:}type, ?t)}{(?t, rdf{:}type, rdfs{:}Class)}$	$\frac{(?u, \dots, \dots) \quad \text{No info on } ?u}{suggest(rdf{:}type)}$ $\frac{(?u, rdf{:}type, \dots) \quad none}{suggestAll(rdf{:}type, rdfs{:}Class)}$
$(P', rdfs{:}domain, T)$ $\forall P = P'$ or P' subproperty of P	$\frac{(?u, P, ?v) \quad none}{(?u, rdf{:}type, T)}$	$\frac{(?\text{u}, \dots, \dots) \quad \text{No info on } ?u}{\text{suggest(P)}}$ $\frac{(?u, \dots, \dots) \quad (?u, rdf{:}type, T)}{suggest(P)}$
$(P', rdfs{:}range, T)$ $\forall P = P'$ or P' subproperty of P	$\frac{(?u, P, ?v) \quad none}{(?v, rdf{:}type, T)}$	$\frac{(?u, P, \dots) \quad none}{suggestAll(rdf{:}type, T)}$
$(P, s{:}domainIncludes, T)$		$\frac{(?u, \dots, \dots) \quad (?u, rdf{:}type, T)}{suggest(P)}$
$(P, s{:}rangeIncludes, T)$		$\frac{(?u, P, \dots) \quad none}{suggestAll(rdf{:}type, T)}$
$(S, rdf{:}type, sh{:}NodeShape)$ and $(S, rdf{:}type, rdfs{:}Class)$	$\frac{none \quad (?u, rdf{:}type, S)}{(?u, :pathsOf, S)}$	
$(S, sh{:}targetNode, U)$	$\frac{none \quad none}{(U, :pathsOf, S)}$	
$(S, sh{:}targetClass, T)$	$\frac{none \quad (?u, rdf{:}type, T)}{(?u, :pathsOf, S)}$	$\frac{(?u, rdf{:}type, \dots) \quad none}{suggest(T)}$
$(S, sh{:}targetSubjectsOf, P)$	$\frac{(?u, P, ?v) \quad none}{(?u, :pathsOf, S)}$	$\frac{(?\text{u}, \dots, \dots) \quad \text{No info on } ?u}{\text{suggest(P)}}$ $\frac{(?u, \dots, \dots) \quad (?u, :pathsOf, S)}{suggest(P)}$ $\frac{(?u, \dots, \dots) \quad none}{suggest(P)}$
$(S, sh{:}targetObjectsOf, P)$	$\frac{(?u, P, ?v) \quad none}{(?v, :pathsOf, S)}$	$\frac{(?u, P, \dots) \quad none}{suggestAll(:pathsOf, \text{S})}$
$(S1, sh{:}node, S2)$ and $S1$ is a node shape	$\frac{none \quad (?u, :pathsOf, S1)}{(?u, :pathsOf, S2)}$	

6.4.2 Rules built from SHACL Paths

Similarly to SPARQL paths, a SHACL path can be either a predicate path (an outgoing triple with a given predicate) or a composition of other paths with one of the following operators: inverse (`sh:inversePath`), sequence, alternative (`sh:alternativePath`), repetition (`sh:oneOrMorePath`), Kleene (`sh:zeroOrMorePath`), and optional (`sh:zeroOrOnePath`).

One issue with paths is that we want to be able to process complex paths, and provide suggestions at any point in the path. For example, for the sequence path `( :a :b )`, `:b` should be a suggested predicate for nodes targeted by `:a`.

Because complex paths exist, we can not simply rely on a set of meta rules for SHACL paths and we must use another solution.

Unit paths and virtual shapes Our solution is to split composite paths into what we consider unit paths. Unit paths are either predicate paths, *e.g.* `:owns`, or inverse paths of a predicate path, *e.g.* `[ sh:inversePath :ownedBy ]`. These unit paths are connected with *virtual shapes*, shapes that do not explicitly exist in the original shape graph. The chain of all the unit paths through the virtual shapes is equivalent to the original composite path for the purpose of our suggestion engine.

Let us consider the shape graph on Listing 6.2. This shape graph means any node `?postalAddress` extracted from the SPARQL request on Listing 6.3 must comply with the node shape `s:PostalAddress`. For our purpose, it is equivalent to the shape graph on Listing 6.4 where we introduced a new shape, `ex:VirtualShape` that acts as the shape of all the matches for `?o` in the SPARQL request.

Listing 6.2: An example shape

```
s:Person rdf:type sh:NodeShape ;
  sh:targetClass s:Person ;
  sh:property [
    sh:path (
      s:worksFor
      s:address
    ) ;
    sh:node s:PostalAddress
  ] .
```

Listing 6.3: SPARQL query to get all the resources targeted by the property shape contained by `s:Person`

```
SELECT ?postalAddress WHERE {
  # All resources targeted by the shape
  ?person rdf:type s:Person .
  # Travel the path
  ?person s:worksFor ?o .
  ?o s:address ?postalAddress .
}
```

Listing 6.4: The same shape graph with a virtual shape

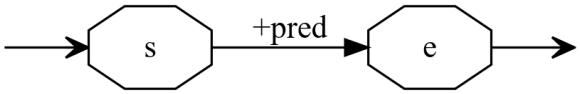
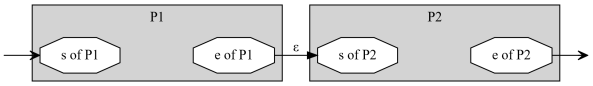
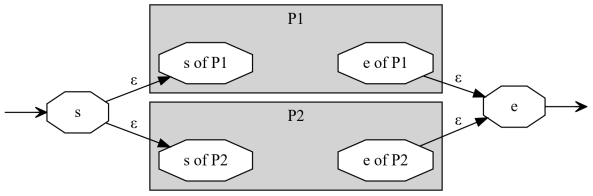
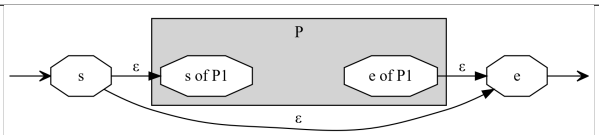
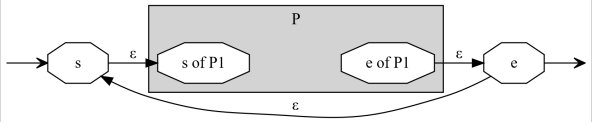
```
s:Person rdf:type sh:NodeShape ;
  sh:targetClass s:Person ;
  sh:property [
    sh:path s:worksFor ;
    sh:node ex:VirtualShape01
  ] .

ex:VirtualShape01
  rdf:type sh:NodeShape ;
  sh:property [
    sh:path s:address ;
    sh:node ex:PostalAddress
  ] .
```

Overview on transforming SHACL Paths into rules. To process SHACL paths, we assume that:

- We can decompose any path into unit paths connecting virtual shapes.
- Processing a chain of unit predicate paths is similar to processing a string with a regex. Hence, we can use finite-state automaton (FSA) to recognize if a chain of triples is recognized by a path.

Table 6.2: Mapping from all kinds of path to automata

Kind and SHACL Syntax	***Regex equivalent***	***Built automaton***
Predicate `pred`	p	
Inverse `[ sh:inversePath P ]`	None	Take automaton P Inverse all transitions Transform all + into - Transform all - into +
Sequence `( P1 P2 )`	P1 P2	
Alternate `[ sh:alternatePath ( P1 P2 ) ]`	(P1 | P2)	
Zero or one `[ sh:zeroOrOnePath P ]`	P?	
One or more `[ sh:oneOrMorePath P ]`	P+	
Zero or more `[ sh:zeroOrMorePath P ]`	P*	Equivalent to (P+)?

- The only difference between a predicate path and an inverse predicate path is whether the corresponding shape applies to the subject or the object.

Based on these assumptions, to parse the SHACL path into a list of inference and suggestion rules, we first transform the path into an FSA, then we transform the FSA into rules.

From SHACL paths to FSA. We build the FSA that describes the path P by composition. Predicate paths produce an FSA with two states and only one transition. The FSA of other paths, that are composite paths, are built by combining the automaton of their components in some way. The transition symbol used by all the produced automata are composed by combining either the sign + for out-going edges or − for incoming edges, with the predicate to travel. Table 6.2 describes all the composition rules, where we consider that pred/p is any predicate, P, P1 and P2 are paths. In the third column, s (start state) and e (end state) are new fresh states that are built each time the automaton is built

Table 6.3: Converting the transitions of the produced FSA to rules

Transition	***Inference rules***	***Suggestion rules***
$(S, +P, E)$	$\frac{(?u, P, ?v) \quad (?u, :pathsOf, m(S))}{(?v, :pathsOf, m(E))}$	$\frac{(?u, \dots, \dots) \quad (?u, :pathsOf, m(S))}{suggest(P)}$ $\frac{(?u, P, \dots) \quad (?u, :pathsOf, m(S))}{suggestAll(:pathsOf, m(E))}$
$(S, -P, E)$	$\frac{(?u, P, ?v) \quad (?v, :pathsOf, m(S))}{(?u, :pathsOf, m(E))}$	

From FSA to rules. After minimization and determination, an FSA can be defined as one initial state, a set of final states and a set of transitions (*StartState*, *Symbol*, *EndState*).

- We define m a total function from all states *state* of the FSA to RDF Nodes. For each state *state*, $m(state)$ is a fresh RDF node, *i.e.* it is not used elsewhere.
- The virtual shape mapped from the initial state of the FSA is a super-shape of the starting shape of the property shape, *i.e.* any resource that complies with the starting shape also complies with the initial state shape.
- If a destination shape is known for the property shape, it is declared as a sub-shape of all the final states of the FSA, *i.e.* any resource that complies with the final state shape also complies with the destination shape of the property.
- Table 6.3 describes how to convert the transitions to inferences rules.

6.5 Inside the Shacled Turtle white box when writing a PRSC context

Using the example exposed in Section 6.1, we are going to see how the Shacled Turtle engine behaves.

The preprocessing Shacled Turtle first requires the user to load a schema graph. Let us consider that the loaded schema graph is the PREC ontology, and in particular the triples exposed in Figure 6.1.

Following the ruleset exposed in Section 6.4, the first rules generated by the preprocessor are the ones described in Table 6.4. Similar rules are produced for `prec:PRSCEdgeRule`, `prec:PropertyRule`, and all types and properties that are not exposed in the listing but exist in the PREC ontology.

The interaction loop We now study the interaction loop through the example exposed in Figure 6.1.

The list of rules triggered by each subsequent user interaction are exposed in Table 6.5.

Table 6.4: The first rules generated from Listing 6.1.

Triple in schema graph	***Inference rules***	***Suggestion rules***
	$\frac{(?u, rdf{:}type, ?t) \quad none}{(?u, rdf{:}type, ?t)}$ $\frac{none \quad (?u, rdf{:}type, ?t)}{(?t, rdf{:}type, rdfs{:}Class)}$	$\frac{(?u, \dots, \dots) \quad \text{No info on } ?u}{suggest(rdf{:}type)}$ $\frac{(?u, rdf{:}type, \dots) \quad none}{suggestAll(rdf{:}type, rdfs{:}Class)}$
$(prec{:}PRSCNodeRule, rdf{:}type, sh{:}NodeShape)$ $(prec{:}PRSCNodeRule, rdf{:}type, rdfs{:}Class)$	$\frac{none \quad (?u, rdf{:}type, prec{:}PRSCNodeRule)}{(?u, {:}pathsOf, prec{:}PRSCNodeRule)}$	
$(prec{:}PRSCNodeRule, sh{:}property, \_{:}n1)$ $(\_{:}n1, sh{:}path, prec{:}label)$		$\frac{(?u, \dots, \dots) \quad (?u, {:}pathsOf, prec{:}PRSCNodeRule)}{suggest(prec{:}label)}$ [3]
$(prec{:}PRSCNodeRule, sh{:}property, \_{:}n2)$ $(\_{:}n2, sh{:}path, prec{:}propertyKey)$		$\frac{(?u, \dots, \dots) \quad (?u, {:}pathsOf, prec{:}PRSCNodeRule)}{suggest(prec{:}propertyKey)}$ [3]
$(prec{:}PRSCNodeRule, sh{:}property, \_{:}n3)$ $(\_{:}n3, sh{:}path, prec{:}produces)$		$\frac{(?u, \dots, \dots) \quad (?u, {:}pathsOf, prec{:}PRSCNodeRule)}{suggest(prec{:}produces)}$ [3]
$(prec{:}PRSCEdgeRule, rdf{:}type, sh{:}NodeShape)$ $(prec{:}PRSCEdgeRule, rdf{:}type, rdfs{:}Class)$	$\frac{none \quad (?u, rdf{:}type, prec{:}PRSCEdgeRule)}{(?u, {:}pathsOf, prec{:}PRSCEdgeRule)}$	
$(prec{:}PRSCEdgeRule, sh{:}property, \_{:}n4)$ $(\_{:}n4, sh{:}path, prec{:}propertyKey)$		$\frac{(?u, \dots, \dots) \quad (?u, {:}pathsOf, prec{:}PRSCEdgeRule)}{suggest(prec{:}propertyKey)}$ [3]
$(\dots)$		
$(prec{:}PropertyRule, rdf{:}type, sh{:}NodeShape)$ $(prec{:}PropertyRule, rdf{:}type, rdfs{:}Class)$	$\frac{none \quad (?u, rdf{:}type, prec{:}PropertyRule)}{(?u, {:}pathsOf, prec{:}PropertyRule)}$	
$(\dots)$		

[3] After determination / minimization

Table 6.5: The rules triggered by the user actions exposed in Figure 6.1.

Input	**Engine**	**Triggered rules**
$\_{:}PersonRule, \dots, \dots)$	Suggestion	$\dfrac{(\_{:}PersonRule, \dots, \dots) \quad \text{No info on } \_{:}PersonRule}{suggest(rdf{:}type)}$
$(\_{:}PersonRule, rdf{:}type, \dots)$	Suggestion	$\dfrac{(?u, rdf{:}type, \dots) \quad none}{suggestAll(rdf{:}type, rdfs{:}Class)}$
$(\_{:}PersonRule, rdf{:}type, prec{:}PRSCNodeRule)$	Inference	$\dfrac{(\_{:}PersonRule, rdf{:}type, prec{:}PRSCNodeRule) \quad none}{(\_{:}PersonRule, rdf{:}type, prec{:}PRSCNodeRule)}$ $\dfrac{none \quad (\_{:}PersonRule, rdf{:}type, prec{:}PRSCNodeRule)}{(\_{:}PersonRule, {:}pathsOf, prec{:}PRSCNodeRule)}$
$(\_{:}PersonRule, \dots, \dots)$	Suggestion	$\dfrac{(\_{:}PersonRule, \dots, \dots) \quad (\_{:}PersonRule, {:}pathsOf, prec{:}PRSCNodeRule)}{suggest(prec{:}label)}$ $\dfrac{(\_{:}PersonRule, \dots, \dots) \quad (\_{:}PersonRule, {:}pathsOf, prec{:}PRSCNodeRule)}{suggest(prec{:}propertyKey)}$ $\dfrac{(\_{:}PersonRule, \dots, \dots) \quad (\_{:}PersonRule, {:}pathsOf, prec{:}PRSCEdgeRule)}{suggest(prec{:}produces)}$

Note that, although the first and the fourth row have the same input, they produce different suggestions due to extra information being added to the meta graph about the `_:PersonRule` resource in the third row.

6.6 Evaluation

Shacled Turtle uses schemas to reduce the number of suggestions proposed to users, keeping only the most relevant ones. The underlying assumption is that this is more helpful for users than a less selective suggestion engine. In order to evaluate the validity of this assumption, we asked volunteers to translate two texts into Turtle documents by using a given ontology. One of the documents had to be written by using our auto-completion engine, the other by using an auto-completion engine similar to the one used by YASGUI, *i.e.* that displays all the terms of the ontology. The order of the two different documents and of the two auto-completion engines was randomized.

We used two different schemas:

- The Schema.org ontology. For this session, we used the RDF schema graph published on Github by Schema.org[4]. We slightly altered the graph to transform the cases where a predicate only had one value for `schema:domainIncludes` or `schema:rangeIncludes` to `rdfs:domain` and `rdfs:range` to help the *inference engine* of Shacled Turtle. This alteration has no impact on the naive suggestion engine. As said previously, Schema.org is a big ontology with thousands of terms. For this session, we had 23 volunteers, 21 of them were Semantic Web experts with more than 3 years of usage and 6 had already used the Schema.org ontology.
- *Friend of a friend* (foaf)[5]. As this ontology is defined by using mostly RDFS, it benefits fully from the *inference engine*. Moreover, it is a small ontology, with a few dozen of terms. For this session, we had 11 volunteers, 7 of them were Semantic Web experts and none declared to already have used the ontology.

Shacled Turtle has been evaluated without any consideration of PREC because 1) requiring users to write PREC contexts would have required them to understand the whole PREC process,

[4]https://github.com/schemaorg/schemaorg/blob/main/data/releases/14.0/schemaorg-all-https.ttl
[5]https://xmlns.com/foaf/spec/

thus reducing the potential pool of users for our experiment, 2) as Shacled Turtle has been designed as a general purpose tool, and not only a tool to be used for PREC, we wanted to evaluate it as such. The produced documents were expected to be constituted of approximately 10 triples.

After writing the two different RDF documents, one with Shacled Turtle and one without it, they were asked to grade on a Likert scale [79] their feeling about the usefulness of both completion engines (naive and Shacled Turtle) and if they preferred an auto-completion engine over another one. We also let users explain in a free field why they preferred one engine, if any; and another free field to collect general feedback. Finally, we measured how much time each volunteer took to write each document.

The whole evaluation was conducted online. We published the source code of the platform and the anonymized collected results on Github at https://github.com/BruJu/shacled-turtle-evaluation.

Of the 34 volunteers, 17 declared to have no preference towards an engine or the other. Six volunteers even admitted having seen no difference between the two engines. The number of people that prefer one engine over another is almost equal for both engines.

When asked separately, all volunteers gave a similar rank to both engines, the worst case being a strong appreciation on an engine and a neutral appreciation on the other; but 21 users gave the same appreciation to both.

Using Shacled Turtle does not enable the user to complete the task faster: 20 volunteers were faster to complete the second task than the first, regardless of if Shacled Turtle is the first engine or the second, and 14 took about the same time.

6.7 General purpose discussion

In this section, we study some of the most recurring comments made by the volunteers about the tool to have a better understanding of what can be improved in Shacled Turtle.

Relevance of suggestions. Five volunteers showed a high enthusiasm about the approach and their comments showed that they understood well the purpose of the tool. In particular, two of them appreciated that the tool leads to fewer errors, feeling more confident about the produced graph.

However, in Section 6.4.1, we mentioned that the choice of which suggestions to filter out is, to some extent, arbitrary, and could lead to false negatives.

The question arises especially in the case of SHACL shapes: we suggest only predicates that are mentioned in the shape(s) of the subject, but unless these shapes are flagged with `sh:closed true`, they actually do not disallow *other* predicates. Similar issues may apply with RDFS classes, because an instance of a class might still be an instance of another one.

Indeed, three volunteers complained about the fact that Shacled Turtle produced fewer suggestions than the other engine. 21 volunteers ranked both engines similarly, and six of them explicitly reported that they did not notice any difference, suggesting that there is no clear benefit in reducing the overall number of suggestions.

Other filtering strategy. Most auto-completion engines enable users to filter the list of suggestions by name. In Code Mirror, and therefore in Shacled Turtle, when the user types for example `s:na`, the system will only show the terms that contain the characters `s:na` in that order (e.g. `s:familyName` or `s:eventStatus`). A common practice to find a desired term is

to opportunistically reduce the list of terms using the filtering by name. Then when the user considers the list of terms to be short enough, they look further at the displayed terms. The responses of the volunteers indicate that they proceeded that way.

Therefore, it might be more valuable to *promote* the suggestions we deem relevant than to filter out the others, and leave it to the user to reduce the number of suggestions using filtering by name. Once a suggestion list is filtered out by the user, we think that Shacled Turtle could provide an efficient strategy to help the user pick the right term, in conjunction with manual filtering by name.

The importance of good documentation. Shacled Turtle shows, with each suggested term, a description (`rdfs:comment`) of that term when provided by the schema. While the query GUI of Wikidata does the same, because Wikidata IRIs are opaque, many other suggestions engine do not. During our experiment, seven volunteers reported that the descriptions of the terms are important, as they complained when descriptions were missing or incomplete, either because of bugs during the early stages of the experiment or because of the used schema. Five volunteers reported to have consulted the ontology online documentation to check how to use the ontology and have a better idea of the usage of the terms and their links. At the opposite, a volunteer reported that thanks to this tool, they fortunately did not feel the necessity to consult the ontology documentation.

One of the volunteers explained that the domain and the range of a property can be more informative than a description. The *schema to rules converter* could also be used to enrich the descriptions to add the links between the predicates and the different types and shapes.

As mentioned previously, Shacled Turtle should not be used to filter out choices from contextual data, but to enrich the documentation. This could be changed by using Shacled Turtle to *promote* terms that we deem relevant, either by displaying them first in the list, by highlighting them, or both: it would solve the issue of users not seeing a difference. To increase the perceived reliability of the tool, the decision made should be explained to the users, *i.e.* in case of an incomplete triple with only a subject, displaying which type or shape of the subject is used to suggest each relevant predicate; and for an incomplete triple with only a missing object, which type or shape of the suggested objects is used to suggest them depending on the subject and the predicate.

6.8 Shacled Turtle and PREC requirements

Unlike most ontologies, the PREC defined types are not intended to be mixed with other ontologies: rules are expected to only use the terms in the ontology, and possibly some generic descriptive predicates such as `rdfs:label`. Rules are also expected to be of one and only one type. The fact that the suggestions of Shacled Turtle are limited is a desired property, as a context using unexpected terms will most of the time be ignored, which might be unexpected from the point of view of the user. Listing 6.5 shows three examples of triples that will be ignored by the PREC engine: the second and third triples will have no effect which may surprise users. On the opposite, the fourth triple being ignored by the PREC engine is probably expected.

Listing 6.5: Some unexpected predicates for a PRSC node rule

```
_:MyRule a prec:PRSCNodeRule ;
  # Will be ignored as it is a PREC-C predicate, this is unexpected by the user
  prec:priority 2 ;
  # This triple will have no effect which is probably unexpected by the user
  foaf:birthdate "birthdate"^^prec:valueOf ;
  # The user probably intended to write
```

```
  #   prec:produces << pvar:self foaf:birthdate "birthdate"^^prec:valueOf >>
  # However, this triple will have the expected behaviour
  rdfs:label "My rule" ;
.
```

However, Shacled Turtle in the context of writing PREC contexts suffers from the lack of RDF-star support. As it is impossible to write SHACL property shapes about an embedded triple, it is impossible to populate the suggestion engine with suggestions about those triples. Writing template graphs could also benefit from a useful auto-completion engine, in particular to suggest the relevant terms in the `pvar` namespace, or to suggest `prec:valueOf` as the datatypes of literals.

6.9 Conclusion

As writing PREC contexts can be a tedious task, especially for newcomers, in this chapter, we tackled the problem of writing RDF documents by hands. For this purpose, we proposed Shacled Turtle, an auto-completion engine that resorts to a schema graph to suggest terms related to the types and shapes of the subject of the triple that the user is writing. The system relies on two different rule engines: an *inference engine* that deduces the list of types and shapes of all resources in the *currently written graph*, and a *suggestion engine* that provides possible following terms. However, in our experiments, the users barely saw any difference between a naive approach, proposing all terms that are in the ontology, and our approach: to find appropriate terms, they preferred to rely on other strategies like filtering by name and reading the ontology online documentation. We explain this by the inability of our method to display explicit insights: the difference between Shacled Turtle and the naive approach is implicit, as it consists in showing less options.

As users are in quest of information, four aspects can be considered:

- Enriching the descriptions of the terms, both with information extracted from the *schema to rules converter* like the links between the predicates and the types and shapes, and with contextual information to explain why the system thinks a term may be relevant in the current *incomplete triple*.
- Instead of using Shacled Turtle to filter out irrelevant terms, promote these relevant terms in the list of all existing terms.
- Running an inference engine to provide the list of types of the resources when the user hovers the resource. While this is currently done for RDFS, it could be expanded to any inference rule-set like OWL.
- Using a SHACL validation report to report errors, *i.e.* as a linting tool. This would lead to more accurate information and more visible error.

Another perspective is to propose snippets, i.e. complete set of triples, instead of simple paths. SHACL sequence paths are paths composed of other paths: instead of requiring the user to chain blank nodes for each path that composes the sequence path, a snippet could be suggested that would build all the intermediate blank nodes at once. This approach would better benefit from SHACL paths and offer a higher level of suggestion.

Finally, the current rule-set does not allow considering RDF-star to be able to provide suggestions for quoted triples. To support RDF-star, we either need to wait for the integration of RDF-star in existing schema languages like SHACL[6], to develop a specific ontology built to be able to express nested terms suggestion in Shacled Turtle, or use another existing ontology that already supports RDF-star and that could be diverted to be used for auto-completion like

ShEx-star[80]. However, to the best of our knowledge, no such ontology currently exists and building a new ontology contradicts the original philosophy of Shacled Turtle to provide an auto-completion tool with already existing ontology documents. As Shacled Turtle is able to guide users through the useful predicates for the rules they chose, it can be seen as the first stepping stone to build a graphical user interface to use PREC.

Acknowledgments

We would like to thank all the volunteers for their time and their very valuable feedback.

[6]Solutions for the integration of RDF-star in SHACL are for example discussed at https://github.com/w3c/shacl/issues/15.

Chapter 7

Conclusion

Studied approaches in this thesis In this thesis, we studied the problem of converting PGs into RDF graphs in a manner that is driven by the user. The purpose was to produce an idiomatic RDF graph, *i.e.* an RDF graph that uses existing ontologies, and/or design patterns that are commonly used in RDF. The user choice is materialized by the fact that the user has to provide to the algorithm two inputs: the PG to transform and a description of how to represent the content of the PG in RDF, named a context. In the course of the thesis, two different algorithms have been proposed: PREC-C and PRSC. For each algorithm, the context contains different information.

The PREC-C algorithm, presented in Chapter 4, relies on a context that maps each NEP (node, edge or property) of the PG, to a template graph. PREC-C is then responsible for instantiating the template graph with the data related to the NEP. The PREC-C approach is an approach where all PGs can be transformed by any context, and the context is a method to override the PREC-C default behavior for certain NEPs, that are specified by a notion of selectors.

The PRSC algorithm, presented in Chapter 5, relies on contexts that contain both a schema and how to translate into RDF the types described by the schema. In PRSC, types are defined as a specific set of labels and properties that a node or edge of that type must have. On the opposite of PREC-C, a PRSC context can only be used to transform PGs that comply with its schema.

Both PREC-C and PRSC require the user to use RDF to express the context. As PREC-C has a large number of different predicates, the question of how to help potential users getting familiar with the PREC-C ontology was raised. We proposed Shacled Turtle in Chapter 6, a method to populate the list of terms proposed by an auto-completion engine when writing Turtle documents. This tool relies on the RDFS and SHACL description of the PREC ontology to populate the list of proposed terms.

Observations on each proposed solution

PREC-C and PRSC PREC-C and PRSC have different advantages and drawbacks. The PREC-C algorithm can work on any PG with any PREC-C context. It lets the user quickly convert a PG to RDF and start exploiting the produced RDF data. Through the use of our GPG (Gremlinable Property Graph) formal definition (Definition 2), it has been designed to cover all existing implementations of PGs that we are aware of. The only constraint is that it can not support a potential PG implementation that allows any number of labels on edges. The PREC-C context enables the user to override a part or the totality of the rules used to

produce the RDF graph by specifying templates. It enables the user to build incrementally their context, by overriding the PREC-0 behavior on parts of the graphs until all selectors use an overridden template graph. However, PREC-C lacks ergonomics for the user. The PREC-C ontology contains a lot of terms, and a lot of specific placeholders are allowed in the template graphs. The concept of overriding the default behavior on a super-set of the specified selectors in the context can be hard to understand. In addition, experience shows that the tool behavior is generally not well understood: some papers citing PREC wrongly report that PREC-C does not support features that it actually supports like generating nested triples. In other words, PREC-C is a very powerful tool to use, especially on properties as it supports any depth of meta-properties. However, it is very difficult to understand and to use.

On the opposite, the PRSC algorithm only works on some PGs that are specified by the schema of the PRSC context. The content of the PG to convert is first checked against any type explicitly specified by the PRSC context, and for each type, the PRSC context specifies a template graph. While PREC-C selectors can either select nodes, edges or properties, the types supported by PRSC are only about nodes and edges: properties are part of the type. By consequence, if a property is optional for a given node label or edge label, PRSC requires specifying a lot of different types: one for each variation of properties. The presented version does not support some common features supported by PGs engines, especially meta-properties and multi-valued properties. Instead, PRSC relies on the very common definition of PGs by Angles and is based on solid theoretical foundations. In particular, a subset of the PRSC contexts named PRSC well-behaved contexts have been studied: for these contexts, we formally proved that the conversion operated by PRSC is reversible, *i.e.* from the produced RDF graph and the used PRSC context, we are able to revert back to the PG that produced it. While PRSC seems less usable to quickly produce an RDF graph from a PG, we proposed a default PRSC context that works for any PG. In addition, it is possible to imagine to first generate a context with all the types present in the PG, and require the user to only write the template graphs for each type.

Overall, thanks to the fact that the PRSC conversion is easier to understand and provide some kind of error checking through the fact that it is based on a schema, the PRSC algorithm should be preferred other the PREC-C algorithm. PREC-C should only be used for edge cases that are not yet supported by PRSC. To enhance PRSC usability, it may be possible to import property selectors from PREC-C to PRSC. It would solve the problem of PRSC being able to only support closed types, *i.e.* types with a given set of properties and no others. It would also help to support some kind of meta-properties for which the semantics is shared across all its usage like a *provenance* meta-property.

Shacled Turtle While Shacled Turtle received good informal feedback on the principle, the user experiments showed disappointing results. This is due to the fact that users rely more on a text filtering strategy, consisting on typing the first letters of the term they want and reading the description when there are few, instead of relying on the fact that the engine is smart enough to remove some predicates based on the type of the resources. In fact, some users were unhappy about the fact that the engine removes some options. User feedback reported that instead of using the schema analysis from Shacled Turtle to remove predicates, the auto-completion engine should instead promote the terms that it considers as relevant, and explain the reasoning behind it. Showing the result of Shacled Turtle in the term descriptions would also contribute to provide more precise descriptions about the terms, by adding for example the domain and the range of a suggested predicate, instead of natural language descriptions that may be too vague. Note that the experiments were performed on general purpose ontologies:

in this context, as they did not find the right predicate in the proposed ontology, some of the volunteers tried to use predicates that are not related to the classes of a term, and others tried to import other ontologies which were intentionally blocked in the experiments; these volunteers expressed frustration that these attempts were not successful. However, in the case of writing mappings, like PREC contexts or RML documents, limiting the creativity of the user may actually be a good thing: any term that is used in an unexpected context will be ignored. For example the `prec:nodeLabelIRI` predicate used on a PREC-C's `prec:EdgeRule` or on a `prec:PRSCNodeRule` will be ignored as it can only be used on PREC-C's `prec:NodeLabelRule`; in these instances, the system should not propose this predicate, and highlight the error if it is used by the user as it would lead to an unexpected behavior from the point of view of the user. As RDFS and SHACL do not support RDF-star, the problem of suggesting relevant terms in quoted triples is still untackled, despite quoted triples being a main component of PREC through how template graphs are expressed. The Shacled Turtle utility can also be discussed with respect to the proposed algorithms: PREC-C strongly requires the user to look at examples or use an auto-completion engine to be usable because there are a lot of terms. On the opposite, PRSC, which should be preferred over PREC-C, requires less a tool such as Shacled Turtle because it uses less than ten terms. Moreover, as it only uses two types, providing an overview of all PRSC capacities through a short example is easier and sufficient.

Template triples In this thesis, we also proposed the concept of template graphs and template triples: instead of using complex constructions to describe the triples to produce like the templating system of R2RML, we use quoted triples to describe the triples to produce with actual RDF triples. While this method provides a very intuitive way to represent RDF triples, it restricts how template placeholders can be encoded. More specifically, literals can only be used in the object position. This is especially annoying as it missed the use-case of producing IRIs that contain the value of some properties: instead of forging IRIs for the NEPs for the PG, the user must stick with blank nodes. The idea behind the OTTR templating system of using lists instead of quoted triples could be a method to express template triples with terms that can be templated. While we proposed a brand-new ontology to express the rules, it could be possible to develop an alternative method to represent PREC contexts, and in particular PRSC contexts, in RML: the type of the PG elements corresponds to the logical source in RML, and the template graph can be expressed through usual RML constructs. However, RML forces the user to use the same subject for all triples generated from a given logical source.

Perspectives In the whole thesis, we only considered transforming all the data contained in a PG into RDF, and studied the reversion in the case of PRSC. However, in a real world context, especially in big data, converting the whole content of the database may not be an acceptable approach because it would take too much time, or lead to data duplication causing in the long term two versions of the data that may drift apart. The main benefit of converting a graph from one model to another is to benefit from the tools of the other graph model: the Cypher or Gremlin API from PGs, shared ontologies from RDF, already standardized validation schemas from RDF, inference systems from RDFS/OWL... The transformation of queries from one model to query the other model is especially a widely studied problem in the literature, but always with a fixed encoding of one model in the other. For example, consider an hypothetical tool that enables to query PGs with SPARQL: if the transformation assumes that PG edges are all represented using RDF reification, the user can not change it, and must use RDF reification to query PG edges. If one wants to use PRSC contexts to change how the content of a PG is represented, and query the PG data using SPARQL, the interaction between contexts and

queries must be studied. For example, if only PG nodes of a certain type use a given predicate in a context, and if the query requires the same predicate, then the engine must be able to use the appropriate Gremlin or Cypher constructs to only query the nodes of this type. However, note that even in the case where no term is specified, querying a triple such as `<< ?ss ?sp ?so >> ?p ?o` already adds some potential constraints about the PG elements that must be queried, for example if only one type of PG element can produce nested triples.

Another aspect left for future work is to study the kind of transformations that are possible to apply on an RDF graph produced from a PRSC well-behaved context that do not hinder the possibility to revert to a valid PG.

On a practical level, the biggest focus of the thesis is on a theoretical level, especially on the study of PRSC contexts. Studying the actual usability of the three proposed solutions is left for future works:

- From our own experience and from the analysis of how other works talk about PREC-C, PREC-C in itself seems to be too complex. Improving PREC-C itself may not be useful, however, reusing the management of properties in PREC-C to improve PRSC may be a good idea to let the latter manage some corner cases like open types (types that allow unexpected properties).
- PRSC has been shown to have good properties, in particular the reversibility of some PRSC contexts has been proven. While the constraints of PRSC well-behaved context, especially with the edge-unique extension, may seem to be reasonable, we still have to check that they are not constraining too much actual users. In addition, we still need to run some user experiments to evaluate how easy and how useful PRSC is for an end-user that wants to convert a PG into an RDF graph. We also studied the theoretical complexity of PRSC and showed that it has a good theoretical complexity: we still have to verify with actual data if the algorithm is able to convert big PGs in a reasonable time. However, as PGs are mostly used in an industry context, big PGs are not freely available to use.
- We propose a first iteration of Shacled Turtle, an auto-completion tool based on schemas. While the first results were disappointing, we received a lot of feedback about the tool and ideas to improve it. The tool could be improved further by implementing term promotion and using schemas to improve the term descriptions; and then running another batch of user experiment. We also did not evaluate the tool in the context of writing PREC contexts or RML mappings, as it would require volunteers to be familiar with these tools, and also familiar with the format of the data to convert. However, these experiments should still be run to confirm or invalidate the intuition that Shacled Turtle is a tool that help the end-users writing these mappings.

Finally, a totally unexplored approach in this thesis is using ontology alignment processes. The idea of the thesis was to help users write a mapping that convert their PG into the RDF graph that they want, and study what are the good properties of these mapping, both in terms of 1) the possibilities provided by the mapping language, *i.e.* the PRSC approach based on schemas seems to be better than the PREC-C approach based on a more low level consideration of NEPs, and 2) in terms of the mappings themselves, *i.e.* well-behaved PRSC contexts have been proved to be reversible. However, another approach, would have been to study how usable ontology alignment techniques are in this context, *i.e.* use an already existing tool to convert data from PG to RDF, for example with PGO, then from the produced RDF graph, transform it by aligning the ontology produced by PGO with existing ontologies.

Bibliography

[1] Heiko Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web*, 8(3):489–508, 2017.

[2] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. *SEMANTiCS (Posters, Demos, SuCCESS)*, 48(1-4):2, 2016.

[3] Edward W Schneider. Course modularization applied: The interface system and its implications for sequence control and data analysis. 1973.

[4] Gavin Carothers and Eric Prud'hommeaux. RDF 1.1 turtle. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-turtle-20140225/.

[5] Ramanathan Guha and Dan Brickley. RDF Schema 1.1. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[6] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[7] Olaf Hartig. Foundations to query labeled property graphs using SPARQL. In *SEM4TRA-AMAR@ SEMANTICS*, 2019.

[8] Nico Baken. Linked data for smart homes: Comparing RDF and labeled property graphs. *LDAC2020*, pages 23–36, 2020.

[9] Davide Alocci, Julien Mariethoz, Oliver Horlacher, Jerven T Bolleman, Matthew P Campbell, and Frederique Lisacek. Property graph vs RDF triple store: A comparison on glycan substructure search. *PloS one*, 10(12):e0144578, 2015.

[10] Souripriya Das, Jagannathan Srinivasan, Matthew Perry, Eugene Inseok Chong, and Jayanta Banerjee. A tale of two graphs: Property graphs as RDF in oracle. In *EDBT*, pages 762–773, 2014.

[11] Ora Lassila, Michael Schmidt, Olaf Hartig, Brad Bebee, Dave Bechberger, Willem Broekema, Ankesh Khandelwal, Kelvin Lawrence, Carlos Manuel Lopez Enriquez, Ronak Sharda, et al. The onegraph vision: Challenges of breaking the graph model lock-in 1. *Semantic Web*, (Preprint):1–10, 2023.

[12] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. Mapping RDF databases to property graph databases. *IEEE Access*, 8:86091–86110, 2020.

[13] Ghislain Auguste Atemezing and Anh Huynh. Knowledge graph publication and browsing using neo4j. In *The 1st workshop on Squaring the circle on graphs*, 2021.

[14] Dominik Tomaszuk, Renzo Angles, and Harsh Thakkar. PGO: Describing property graphs in RDF. *IEEE Access 8*, pages 118355–118369, 2020.

[15] Xiaohan Zou. A survey on application of knowledge graph. In *Journal of Physics: Conference Series*, volume 1487, page 012016. IOP Publishing, 2020.

[16] Dieter Fensel, Umutcan Şimşek, Kevin Angele, Elwin Huaman, Elias Kärle, Oleksandra Panasiuk, Ioan Toma, Jürgen Umbrich, Alexander Wahler, Dieter Fensel, et al. Introduction: what is a knowledge graph? *Knowledge graphs: Methodology, tools and selected use cases*, pages 1–10, 2020.

[17] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[18] Seema Sundara, Richard Cyganiak, and Souripriya Das. R2RML: RDB to RDF mapping language. W3C recommendation, W3C, September 2012. https://www.w3.org/TR/2012/REC-r2rml-20120927/.

[19] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. RML: a generic language for integrated RDF mappings of heterogeneous data. In *Ldow*, 2014.

[20] Pierre-Antoine Champin, Gregg Kellogg, and Dave Longley. JSON-ld 1.1. W3C recommendation, W3C, July 2020. https://www.w3.org/TR/2020/REC-json-ld11-20200716/.

[21] Renzo Angles. The property graph database model. In *AMW*, 2018.

[22] Jürgen Hölsch and Michael Grossniklaus. An algebra and equivalences to transform graph patterns in Neo4j. 2016.

[23] Giacomo Bergami. On efficiently equi-joining graphs. In *Proceedings of the 25th International Database Engineering & Applications Symposium*, pages 222–231, 2021.

[24] Martin Junghanns, André Petermann, Niklas Teichmann, Kevin Gómez, and Erhard Rahm. Analyzing extended property graphs with apache flink. In *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics*, pages 1–8, 2016.

[25] Martin Junghanns, André Petermann, and Erhard Rahm. Distributed grouping of property graphs with GRADOOP. 2017.

[26] Giacomo Bergami, Matteo Magnani, and Danilo Montesi. A join operator for property graphs. In *EDBT/ICDT Workshops*, 2017.

[27] József Marton, Gábor Szárnyas, and Dániel Varró. Formalising opencypher graph queries in relational algebra. In *Advances in Databases and Information Systems: 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings 21*, pages 182–196. Springer, 2017.

[28] Harsh Thakkar, Dharmen Punjani, Sören Auer, and Maria-Esther Vidal. Towards an integrated graph algebra for graph pattern matching with gremlin. In *Database and Expert Systems Applications: 28th International Conference, DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part I 28*, pages 81–91. Springer, 2017.

[29] Ora Lassila. Resource description framework (RDF) model and syntax specification. W3C recommendation, W3C, February 1999. https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[30] Frank Manola, Eric Miller, Brian McBride, et al. RDF primer. *W3C recommendation*, 10(1-107):6, 2004.

[31] Guus Schreiber and Yves Raimond. RDF 1.1 primer. W3C note, W3C, June 2014. https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[32] Olaf Hartig, Pierre-Antoine Champin, Gregg Kellogg, and Andy Seaborne. RDF-star and SPARQL-star. *W3C Community Group Report, Online at https://www. w3. org/2021/12/rdf-star. html*, 2021.

[33] Peter Patel-Schneider and Patrick Hayes. RDF 1.1 semantics. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf11-mt-20140225/.

[34] Steven Harris and Andy Seaborne. SPARQL 1.1 query language. W3C recommendation, W3C, March 2013. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[35] Dimitris Kontokostas and Holger Knublauch. Shapes constraint language (SHACL). W3C recommendation, W3C, July 2017. https://www.w3.org/TR/2017/REC-shacl-20170720/.

[36] Guus Schreiber and Fabien Gandon. RDF 1.1 XML syntax. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/.

[37] Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[38] Dan Brickley and Ramanathan Guha. RDF Schema 1.1. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[39] Lushan Han, Tim Finin, Cynthia Parr, Joel Sachs, and Anupam Joshi. RDF123: from spreadsheets to RDF. In *International Semantic Web Conference*, pages 451–466. Springer, 2008.

[40] Martin G Skjæveland, Henrik Forssell, Johan W Klüwer, Daniel Lupp, Evgenij Thorstensen, and Arild Waaler. Pattern-based ontology design and instantiation with reasonable ontology templates. *A Higher-Level View of Ontological Modeling*, 69, 2019.

[41] Maxime Lefrançois, Antoine Zimmermann, and Noorani Bakerally. A SPARQL extension for generating RDF from heterogeneous formats. In *The Semantic Web: 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28–June 1, 2017, Proceedings, Part I 14*, pages 35–50. Springer, 2017.

[42] Sumit Purohit, Nhuy Van, and George Chin. Semantic property graph for scalable knowledge graph analytics. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 2672–2677. IEEE, 2021.

[43] Paul Warren and Paul Mulholland. Edge labelled graphs and property graphs; a comparison from the user perspective. *arXiv preprint arXiv:2204.06277*, 2022.

[44] Renzo Angles, Aidan Hogan, Ora Lassila, Carlos Rojas, Daniel Schwabe, Pedro A Szekely, and Domagoj Vrgoc. Multilayer graphs: a unified data model for graph databases. In *GRADES-NDA@ SIGMOD*, pages 11–1, 2022.

[45] Ran Zhang, Pengkai Liu, Xiefan Guo, Sizhuo Li, and Xin Wang. A unified relational storage scheme for RDF and property graphs. In *International Conference on Web Information Systems and Applications*, pages 418–429. Springer, 2019.

[46] Vinh Nguyen, Hong Yung Yip, Harsh Thakkar, Qingliang Li, Evan Bolton, and Olivier Bodenreider. Singleton property graph: Adding a semantic web abstraction layer to graph databases. In *BlockSW/CKG@ ISWC*, pages 1–13, 2019.

[47] Fabrizio Orlandi, Damien Graux, and Declan O'Sullivan. Benchmarking RDF metadata representations: Reification, singleton property and RDF. In *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*, pages 233–240. IEEE, 2021.

[48] Olaf Hartig and Bryan Thompson. Foundations of an alternative approach to reification in RDF. *arXiv preprint arXiv:1406.3399*, 2014.

[49] Olaf 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.*, volume 1912. Juan Reutter, Divesh Srivastava, 2017.

[50] Shahrzad Khayatbashi, Sebastián Ferrada, and Olaf Hartig. Converting property graphs to RDF: a preliminary study of the practical impact of different mappings. In *GRADES-NDA@ SIGMOD*, pages 10–1, 2022.

[51] Harsh Thakkar, Dharmen Punjani, Jens Lehmann, and Sören 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)*, pages 1–5, 2018.

[52] Sumit Neelam, Udit Sharma, Sumit Bhatia, Hima Karanam, Ankita Likhyani, Ibrahim Abdelaziz, Achille Fokoue, and LV Subramaniam. Expressive reasoning graph store: A unified framework for managing RDF and property graph databases. *arXiv preprint arXiv:2209.05828*, 2022.

[53] Naglaa Fathy, Walaa Gad, Nagwa Badr, and Mohamed Hashem. Progomap: Automatic generation of mappings from property graphs to ontologies. *IEEE Access*, 9:113100–113116, 2021.

[54] Franck Michel, Loïc Djimenou, Catherine Faron Zucker, and Johan Montagnat. *xR2RML: Relational and non-relational databases to RDF mapping language*. PhD thesis, CNRS, 2017.

[55] Ghadeer Abuoda, Daniele Dell'Aglio, Arthur Keen, and Katja Hose. Transforming RDF-star to property graphs: A preliminary analysis of transformation approaches–extended version. *arXiv preprint arXiv:2210.05781*, 2022.

[56] Hirokazu Chiba, Ryota Yamanaka, and Shota Matsumoto. G2gml: Graph to graph mapping language for bridging RDF and property graphs. In *International Semantic Web Conference*, pages 160–175. Springer, 2020.

[57] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, et al. Pg-schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data*, 1(2):1–25, 2023.

[58] Angela Bonifati, Stefania Dumbrava, Emile Martinez, Fatemeh Ghasemi, Malo Jaffré, Pacôme Luton, and Thomas Pickles. Discopg: property graph schema discovery and exploration. *Proceedings of the VLDB Endowment*, 15(12):3654–3657, 2022.

[59] Nimo Beeren. Designing a visual tool for property graph schema extraction and refinement: An expert study. *arXiv preprint arXiv:2201.03643*, 2022.

[60] Mark A Musen. The protégé project: a look back and a look forward. *AI matters*, 1(4):4–12, 2015.

[61] Jesse Wright, Sergio José Rodríguez Méndez, Armin Haller, Kerry Taylor, and Pouya G Omran. Schímatos: a shacl-based web-form generator for knowledge graph editing. In *International Semantic Web Conference*, pages 65–80. Springer, 2020.

[62] Jonas Kjær Rask, Frederik Palludan Madsen, Nick Battle, Hugo Daniel Macedo, and Peter Gorm Larsen. The specification language server protocol: A proposal for standardised lsp extensions. *arXiv preprint arXiv:2108.02961*, 2021.

[63] Karima Rafes, Serge Abiteboul, Sarah Cohen-Boulakia, and Bastien Rance. Designing scientific SPARQL queries using autocompletion by snippets. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 234–244. IEEE, 2018.

[64] Laurens Rietveld and Rinke Hoekstra. YASGUI: not just another SPARQL client. In *Extended Semantic Web Conference*, pages 78–86. Springer, 2013.

[65] Gergő Gombos and Attila Kiss. SPARQL query writing with recommendations based on datasets. In *International Conference on Human Interface and the Management of Information*, pages 310–319. Springer, 2014.

[66] Sébastien Ferré. Sparklis: An expressive query builder for SPARQL endpoints with guidance in natural language. *Semantic Web*, 8(3):405–418, 2017.

[67] Gabriel de la Parra and Aidan Hogan. Fast approximate autocompletion for SPARQL query builders. 2021.

[68] Thomas Francart. Sparnatural: a visual knowledge graph exploration tool. In *European Semantic Web Conference*, pages 11–15. Springer, 2023.

[69] Bryan Kong Win Chang, Marie Lefevre, Nathalie Guin, and Pierre-Antoine Champin. SPARE-LNC : un langage naturel contrôlé pour l'interrogation de traces d'interactions stockées dans une base RDF. In *IC2015*, Rennes, France, June 2015. AFIA.

[70] Oskar Van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2016.

[71] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pages 1433–1445, 2018.

[72] Richard Cyganiak, David Wood, Markus Lanthaler, Graham Klyne, Jeremy J Carroll, and Brian McBride. RDF 1.1 concepts and abstract syntax. *W3C recommendation*, 25(02):1–22, 2014.

[73] Richard Cyganiak, Seema Sundara, and Souripriya Das. R2RML: RDB to RDF mapping language. W3C recommendation, W3C, September 2012. https://www.w3.org/TR/2012/REC-r2rml-20120927/.

[74] Maxime Lefrançois, Antoine Zimmermann, and Noorani Bakerally. Flexible rdf generation from rdf and heterogeneous data sources with SPARQL-generate. In *European Knowledge Acquisition Workshop*, pages 131–135. Springer, 2016.

[75] Patrick Hayes. RDF semantics. W3C recommendation, W3C, February 2004. https://www.w3.org/TR/2004/REC-rdf-mt-20040210/.

[76] Henry Thompson, Sandy Gao, David Peterson, Ashok Malhotra, Michael Sperberg-McQueen, and Paul V. Biron. W3C xml schema definition language (XSD) 1.1 part 2: Datatypes. W3C recommendation, W3C, April 2012. https://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/.

[77] Gregg Kellogg, Pierre-Antoine Champin, and Dave Longley. Json-ld 1.1–a json-based serialization for linked data. W3C recommendation, W3C, 2020.

[78] Julian Bruyat, Pierre-Antoine Champin, Lionel Médini, and Frederique Laforest. Shacled turtle: Schema-based turtle auto-completion. In *Workshop on Visualization and Interaction for Ontologies and Linked Data 2022, co-located with the International Semantic Web Conference 2022*, volume 3253, pages 2–15, 2022.

[79] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[80] Jose Emilio Labra-Gayo. Extending shape expressions for different types of knowledge graphs. In *1st Workshop on Data Quality meets Machine Learning and Knowledge Graphs, DQMLKG, part of Extended Semantic Web Conference 2024, ESWC24*, May 2024.

Appendices

Appendix A

β redefinition in PREC-C

In Chapter 4, the PREC-C algorithm is presented in an iterative manner. However, after defining the *substitute* function, it is possible to redefine the β function for the final version of the algorithm in Section 4.1.5.

However, attentive readers may have noticed that the β function definition could be rewritten by using the *substitute* function.

Definition 44 [β alternative version]
For a given usage of the function $\beta(t, pg, x)$, let $value$ be the value such that $(key, value) = propdetails_{pg}(x)$

$$\begin{aligned}\beta(t, pg, x) = & substitute(substitute(substitute(substitute(substitute(t,\\ & (?self, x)),\\ & (?source, src_{pg}(x))),\\ & (?destination, dest_{pg}(x))),\\ & (?holder, getHolder(x, pg))),\\ & (?value, value))\end{aligned}$$

Note that this rewriting is possible because the placeholders are never replaced by another placeholder. It also means that the order in which the *substitute* functions are called does not change the β function behavior.

Appendix B

Proof of properties on Property Graphs

In this section, we expose the proof for Theorem 5

B.1 Extra mathematical elements

Definition 45 [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. In other words, the restriction of a function by a set X is equal to a function in which the domain is restricted to the elements of the set X.

Remark 24
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 25
A functional definition of Definition 45 would be, for all functions f, $f|_X : x \mapsto f(x)$ if $x \in X \cap Dom(f)$, undefined otherwise.

Lemma 8
For all functions f, for all sets X_1 and X_2, the union of the function restricted by the two sets is equal to the function restricted by the union of the two sets: $f|_{X_1} \cup f|_{X_2} = f|_{X_1 \cup X_2}$.

Proof.

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

□

Remark 26
For all function f, for all sets X_1 and X_2, $f|_{X_1}$ and $f|_{X_2}$ are always compatible.

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

Proof.

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

□

B.2 Redefinition of the projection

Remark 27
$src_{\pi_m(pg)}$, $dest_{\pi_m(pg)}$ and $properties_{\pi_m(pg)}$ can be redefined by using the restriction:

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

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

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

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

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

□

B.3 Proof of Theorem 5

Remark 28
The property graphs used in the following proof are described with formula. To help readability, for a given PG x, we allow ourselves to use the notation $N(x)$ instead of N_x. Similar notation will be used for $E(x)$, $src(x)$, $dest(x)$, $labels(x)$ and $properties(x)$. For example, $N_{\bigoplus_{m\in N_{pg}\cup E_{pg}}\pi_m(pg)}$ will instead of noted $N(\bigoplus_{m\in N_{pg}\cup E_{pg}}\pi_m(pg))$.

Proof. We first need to check if we can apply the $\oplus$ operator, i.e. if the three conditions of Definition 40 are met:

- When the π function is applied, nodes remain nodes and edges remain edges. The $\oplus$ operator also conserves this property. As $\forall m, N_{\pi_m(pg)} \subseteq N_{pg}$ and $E_{\pi_m(pg)} \subseteq E_{pg}$, the first condition is met.
- The definition of π (restriction of the original function), the definition of $\oplus$ (union of the functions) and the Lemma 8 (the union of two restriction is a restriction) imply that the *src*, *dest* and *properties* are compatible.

As $\oplus$ is commutative and associative, we can write the following decomposition:

$$\bigoplus_{m\in N_{pg}\cup E_{pg}} \pi_m(pg) = (\bigoplus_{m\in N_{pg}} \pi_m(pg)) \oplus (\bigoplus_{m\in E_{pg}} \pi_m(pg))$$

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

Edges (E_{pg}):

$$\begin{aligned}
& E(\bigoplus_{m\in N_{pg}\cup E_{pg}} \pi_m(pg)) = \bigcup_{m\in N_{pg}\cup E_{pg}} E(\pi_m(pg)) && \text{[Definition of } \oplus \text{ on E]}\\
=\ & (\bigcup_{m\in N_{pg}} E(\pi_m(pg))) \cup (\bigcup_{m\in E_{pg}} E(\pi_m(pg)))\\
=\ & (\bigcup_{m\in N_{pg}} \emptyset) \cup (\bigcup_{m\in E_{pg}} \{m\}) = \bigcup_{m\in E_{pg}} \{m\} && \text{[Definition of } \pi \text{ on E]}\\
=\ & E_{pg}
\end{aligned}$$

Nodes (N_{pg}):

$$\begin{aligned}
& N(\bigoplus_{m\in N_{pg}\cup E_{pg}} \pi_m(pg))\\
=\ & (\bigcup_{m\in N_{pg}} N(\pi_m(pg))) \cup (\bigcup_{m\in E_{pg}} N(\pi_m(pg)))\\
=\ & N_{pg} \cup (\bigcup_{m\in E_{pg}} N(\pi_m(pg)))
\end{aligned}$$

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

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

Source of the edges (src_{pg}):

$$
\begin{aligned}
& src(\bigoplus_{m \in N_{pg} \cup E_{pg}} \pi_m(pg)) \\
= & \bigcup_{m \in N_{pg} \cup E_{pg}} src_{pg}|_{\{m\}} \\
= & \ src_{pg} \qquad \text{[per Theorem 8, since } \bigcup_{m \in N_{pg} \cup E_{pg}} \{m\} \supseteq E_{pg} = Dom(src_{pg})]
\end{aligned}
$$

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

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

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

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

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

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

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

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

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

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

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

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

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

□

TH1110_BRUYAT Julian_Manuscrit

9% Suspicious texts

4% Similarities
0% similarities between quotation marks
0% among the sources mentioned
6% Unrecognized languages

Document name: TH1110_BRUYAT Julian_Manuscrit.pdf
Document ID: 9932498db9cc0de7e39a93a1930a8f69336760bc
Original document size: 1.75 MB

Submitter: Mickael Lallart
Submission date: 4/9/2024
Upload type: interface
analysis end date: 4/9/2024

Number of words: 62,767
Number of characters: 363,595

Location of similarities in the document:

Main sources detected

No.	Description	Similarities	Locations	Additional information
1	**TH1093_JEONG Jihyuk_Manuscrit.pdf** | TH1093_JEONG Jihyuk_Manuscrit #aa721a The document is from my document database 94 similar sources	2%		Identical words: 2% (2,534 words)
2	**ZHANG Ruochen_Manuscrit_réduit.pdf** | ZHANG Ruochen_Manuscrit_réduit #235732 The document is from my document database 93 similar sources	2%		Identical words: 2% (2,207 words)
3	**TH1099_FORTIER Patrik_Manuscrit.pdf** | TH1099_FORTIER Patrik_Manuscrit #7c44f8 The document is from my document database 91 similar sources	2%		Identical words: 2% (2,047 words)
4	**TH0908_Lucas Ollivier-Lamarque_Manuscrit.pdf** | TH0908_Lucas Ollivier-L... #ebdcc8 The document is from my document database 90 similar sources	2%		Identical words: 2% (2,031 words)
5	**bonndoc.ulb.uni-bonn.de** https://bonndoc.ulb.uni-bonn.de/xmlui/bitstream/20.500.11811/9083/1/6224.pdf 110 similar sources	2%		Identical words: 2% (2,270 words)

Sources with incidental similarities

No.	Description	Similarities	Locations	Additional information
1	**liris.cnrs.fr** | Angela Bonifati | Laboratoire d'InfoRmatique en Image et Systèmes d'... https://liris.cnrs.fr/page-membre/angela-bonifati	< 1%		Identical words: < 1% (38 words)
2	**www.emse.fr** https://www.emse.fr/~zimmermann/Papers/ekaw2016demo.pdf	< 1%		Identical words: < 1% (39 words)
3	**sferrada.com** https://sferrada.com/uploads/resume.pdf	< 1%		Identical words: < 1% (36 words)
4	**ceur-ws.org** https://ceur-ws.org/Vol-3262/paper12.pdf	< 1%		Identical words: < 1% (35 words)
5	**link.springer.com** https://link.springer.com/content/pdf/10.1007/978-3-030-33220-4_8.pdf	< 1%		Identical words: < 1% (39 words)

Ignored sources

These sources have been excluded by the document owner from the calculation of the similarity percentage.

No.	Description	Similarities	Locations	Additional information
1	**www.semantic-web-journal.net** https://www.semantic-web-journal.net/system/files/swj3675.pdf	22%		Identical words: 22% (14,600 words)
2	**www.semantic-web-journal.net** https://www.semantic-web-journal.net/system/files/swj3426.pdf	8%		Identical words: 8% (5,076 words)

Referenced sources (without similarities detected)

These sources were cited in the paper without finding any similarities.

1 https://www.edchimie-lyon.fr
2 http://e2m2.universite-lyon.fr
3 http://ediss.universite-lyon.fr
4 http://ed34.universite-lyon.fr
5 https://edeea.universite-lyon.fr

FOLIO ADMINISTRATIF

THESE DE L'INSA LYON, MEMBRE DE L'UNIVERSITE DE LYON

NOM : **BRUYAT** DATE de SOUTENANCE : **3 Juin 2024**

Prénoms : **Julian**

TITRE : **Des graphes de propriétés aux graphes de connaisances**

NATURE : **Doctorat** Numéro d'ordre : **2024ISAL0044**

École Doctorale : **École Doctorale en Informatiques et Mathématiques de Lyon**

Spécialité : **Informatique**

RÉSUMÉ :
Les graphes de propriétés et les graphes RDF sont deux familles populaires de base de données graphe. Néanmoins, malgré le fait qu'elles soient toutes les deux basées sur la notion de graphe, ces deux familles ne sont pas interopérables. Les graphes de propriétés sont une famille d'implémentations de base de données très flexible, où des propriétés peuvent être rattachées aux noeuds et aux arcs du graphe. La seconde est un modèle standardisé de description de connaissances, reposant sur des vocabulaires partagés entre tous les graphes RDF. Dans cette thèse, nous définissons des méthodes pour permettre une interopérabilité sémantique entre graphes de propriétés et graphes RDF configurée à travers un "contexte" fourni par l'utilisateur. La première méthode est une méthode bas niveau, compatible avec n'importe quel graphe de propriétés. La seconde méthode est une méthode haut niveau, reposant sur la notion de schéma de graphe de propriétés, et pour laquelle la réversibilité de certains contextes est étudiée formellement. Enfin, pour faciliter l'écriture des "contextes" en RDF, et plus généralement de n'importe quel document RDF, nous proposons une méthode d'auto-complétion basée sur les vocabulaires de schémas RDF existants.

MOTS-CLÉS : **Graphes de propriétés, RDF, Interopérabilité, Autocomplétion, Schéma**

Laboratoire(s) de recherche : **Laboratoire d'Informatique en Image et Systèmes d'information**

Directeur de thèse : **Frédérique LAFOREST**
Co-encadrants : **Pierre-Antoine CHAMPIN, Lionel MÉDINI**

Président du Jury : **Philippe LAMARRE**

Composition du Jury :

Catherine FARON
Olaf HARTIG
Anastasia DIMOU
Jose Emilio LABRA GAYO
Philippe LAMARRE
Frédérique LAFOREST
Pierre-Antoine CHAMPIN
Lionel MÉDINI