Transforming Shape Schemas with Composable Property-Graph Queries

Philipp Seifer
The Software Languages Team, University of Koblenz, Germany

Daniel Hernández
Institute for Artificial Intelligence, University of Stuttgart, Germany

Ralf Lämmel
The Software Languages Team, University of Koblenz, Germany

Steffen Staab
Institute for Artificial Intelligence, University of Stuttgart, Germany
Web and Internet Science Research Group, University of Southampton, UK

Abstract

Property graphs may be constrained by schemas that inform both query engines and human users about the shape of valid data, enforcing a contract between data provider and consumer. Composable property-graph queries transform input graphs into output graphs. Then, the question arises of which schema can be expected after one (or several) transformation steps. We investigate how schema constraints can be inferred given an input schema and a transforming query. Specifically, we propose a reasoning procedure that, given an input schema in ProGS and a query in G-CORE infers an output schema. Since graph updates will happen frequently, our inference procedure does not rely on graph instances, such that the computed output schema applies to all graphs originating from any input graph complying with the input schema. Related work has addressed this problem for SPARQL `CONSTRUCT` queries, encoding it in Description Logics (DLs) so that the output schema is entailed by axioms inferred from input schema and queries. Property graphs and their queries, however, complicate the matter, as property graphs feature label and property annotations as well as first-class edges. Thus, reification has to be used in one way or another, though available DLs lack the means to encode such features directly. We approach this novel challenge via a family of mappings for i) property graphs reified in RDF, aligned with ii) a mapping from ProGS to SHACL and iii) a mapping from G-CORE to SPARQL `CONSTRUCT` queries. In this manner, schema inference for property graphs becomes manageable, as we break apart the problem through the extra mapping layer and utilize efficient DL reasoners. We develop the metatheory regarding the soundness of inferred schema constraints and the semantic equivalence of mapped schemas and queries.

2012 ACM Subject Classification Information systems → Graph-based database models; Information systems → Resource Description Framework (RDF); Information systems → Query languages; Information systems → Extraction, transformation and loading

Keywords and phrases Property Graphs, Validation, Schema Inference, Reification, G-CORE, SPARQL

Digital Object Identifier 10.4230/TGDK.4.2.3

Category Research

Related Version *Extended Version*: `http://arxiv.org/abs/2606.14309`

Supplementary Material *Software (Source Code)*: `https://github.com/softlang/s2s` [40]
archived at `swh:1:dir:a38354c2cab3efb8abdc9e80f611d19aa3531d82`

Funding *Daniel Hernández*: Deutsche Forschungsgemeinschaft (DFG) – SFB 1574 – 471687386.

Received 2025-07-01 **Accepted** 2026-04-03 **Published** 2026-09-03

Editors Stefania Dumbrava, George Fletcher, Matteo Lissandrini, and Riccardo Tommasini

Special Issue Data Management for (Knowledge) Graphs

Generative AI Declaration Generative AI was not used in the preparation of this article.

1 Introduction

Composable property-graph queries transform graphs into graphs. In this paper, we are interested in schema inference for the results of such composable queries. For RDF graphs, the W3C proposes a composable query language, namely SPARQL [18] with its `CONSTRUCT` [27] clause. For *property graphs* (PGs), which we aim at in this paper, the recent GQL [21] standard is not composable (as yet [23]). Accordingly, we leverage G-CORE [3], an earlier *composable* query language for property graphs that inspired GQL.

Property-graph schemas are used to constrain PGs; the constraints can be enforced by validation. There are schema languages, sometimes also called *shape* languages, for PGs, such as the *Property Graph Shapes Language* (ProGS) [44] or PG-Schema [4]. As far as schema constraints of the inference approach in this paper are concerned, both ProGS and PG-Schema are viable candidates. Pragmatically, we commit to ProGS, as it is more directly aligned with the established schema language for RDF, i.e. SHACL [26], which is readily covered by the foundation we rely on. In particular, the correspondence between SHACL and description logics (DL) is well understood [41, 8]. Thus, we can leverage efficient reasoner implementations for inference. We follow the terminology of ProGS and SHACL and refer to schemas as *shapes*.

We refer to shapes validating all possible result graphs of a query as *output shapes*. Similarly, we use the term *input shapes* to refer to the set of shapes over the input graphs of a query. The inference of output shapes is a challenging aspect of the composable query notion: input shapes do not carry over directly to the output but do indeed impact it. Additionally, to obtain sound shapes for all possible query results, inference must rely on the *input shapes* and the *query* as inputs rather than individual result graphs. Access to these statically inferred shapes facilitates numerous use cases, even when considering compositions of multiple queries: they enable tools such as type systems for language-embedded queries (compare [30]) among others (e.g., [50, 35]) that rely on shapes. For query pipelines that are executed repeatedly, such shapes need only be inferred once, which is particularly useful in cases where some level of manual review or selection of shapes is desired. Finally, even in data transformation or integration use cases, where result graphs are stored in databases, they complement shapes inferred from graph instances and can provide insights to developers without query execution.

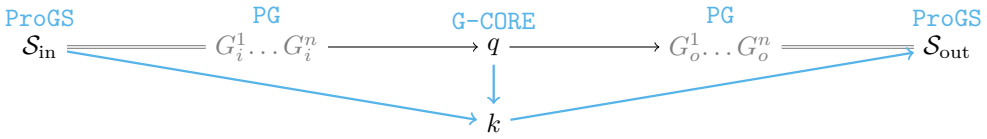


Figure 1 Schematic overview of the problem: Input shapes $\mathcal{S}_{\text{in}}$ validate a (possibly infinite) set of input graphs $G_i^1 \dots G_i^n$ which are valid inputs for the query q. The query produces, as its result, a set of corresponding result graphs $G_o^1 \dots G_o^n$. Just as $\mathcal{S}_{\text{in}}$ validates all input graphs, $\mathcal{S}_{\text{out}}$ validates all output graphs. The set of $\mathcal{S}_{\text{out}}$ is constructed from some formalization k, which is derived from $\mathcal{S}_{\text{in}}$ and q.

Figure 1 shows the input (shapes $\mathcal{S}_{\text{in}}$ and query q) and the output (shapes $\mathcal{S}_{\text{out}}$) of shape inference, with the respective languages annotated in blue. For these languages, we will later define the relevant subsets we consider in this work. The figure also hints in gray at the set of input graphs (conforming to shapes $\mathcal{S}_{\text{in}}$, indicated by gray double lines) and corresponding output graphs (conforming to $\mathcal{S}_{\text{out}}$), with black arrows representing input and output of query execution. These graphs are included in the figure only for the sake of clarity; our approach does not depend on any concrete graphs. Our approach is rendered using blue arrows: we encode in k the information that allows us to deduce which shapes are in $\mathcal{S}_{\text{out}}$.

To introduce k, we establish a common language that can express the constraints of the input shapes $\mathcal{S}_{\text{in}}$ and how inputs and outputs relate to each other over the query q. Given such a language, we then define an algorithm for constructing k from both $\mathcal{S}_{\text{in}}$ and the query q and for inferring the new shapes in $\mathcal{S}_{\text{out}}$ from k.

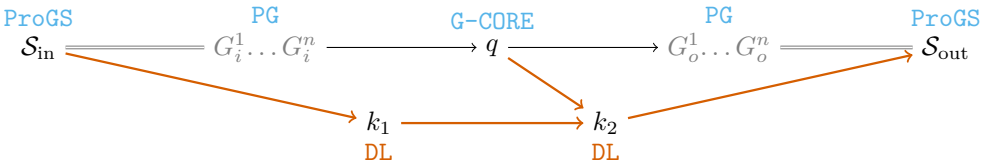


Figure 2 Schematic overview of a possible solution where constraints are encoded in description logics.

One possibility for encoding these constraints is *description logics* (DL). Figure 2 indicates where DL is used with red arrows: we first encode the input shapes $\mathcal{S}_{\text{in}}$ as a set of DL axioms and then combine them with a second set of axioms inferred from the query q. This approach benefits from relying on an existing formalism and, in practical applications, from established and efficient reasoners for implementing the required decision procedures. In prior work [41], we also relied on a DL to solve a similar inference problem for RDF graphs and SPARQL `CONSTRUCT` queries. However, there is a drawback: Common DLs do not support properties with key-value annotations. Thus, we would have to either define a new variant – for which no efficient reasoners exist – or we have to use some form of reification to encode properties in an existing DL.

Since the first option is not desirable, we choose the second. However, instead of encoding reification directly in DL, we introduce an intermediate RDF layer. This provides the following benefits: foremost, it clearly separates the *reification* from the *DL encoding and inference* steps of our solution, making each part easier to formalize, prove, and implement. Secondly, we can build on the aforementioned prior work for the encoding and inference step; we still need to modify and extend the algorithm. Finally, a mapping between the involved shape and query languages has not been formally explored, which we achieve in this work.

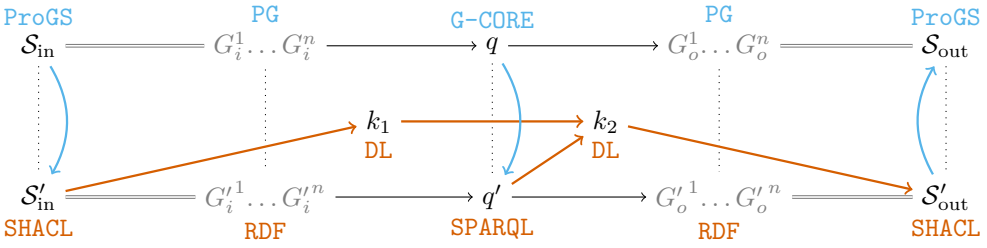


Figure 3 Schematic overview of the involved languages, mappings, and algorithms. The upper PG layer is mapped to the bottom RDF layer, where individual mappings are indicated using dotted lines. Inference and reasoning exclusively operate on RDF. The blue arrows show how mappings are utilized.

Figure 3 shows the full solution we propose. Utilizing reification (indicated with dotted lines) at all levels allows us to step down from a PG-based language family (upper half of Figure 3) to an RDF-based one (lower half of Figure 3). We use RDF for representing the data by reifying first-class edges with identities and properties in PG as ordinary RDF triples. We use SHACL as the shape language, accounting for this reification. This allows us to rely on DL for the formalization and implementation of shape inference, since round tripping between DL axioms and SHACL shapes is possible (cf. [41, 8]). Finally, we use SPARQL `CONSTRUCT` as the composable query language to which we map. Here, the definition of a sound mapping is more complex, since in addition to the reification, we must also account for the differences in query language semantics:

where labels and properties are copied by default in G-CORE and explicitly removed, SPARQL relies on explicit triple patterns and filters. The data reification, shape-to-DL mapping, and shape inference approaches have been studied [24, 4, 41], whereas the query mapping has not.

We build on and extend our existing algorithm [41] that infers SHACL shapes for results of SPARQL `CONSTRUCT` queries on RDF graphs by constructing a DL knowledge base entailing valid output shapes. To define our query mapping, we must extend the SPARQL fragment considered in [41] with more *generic* triple patterns where variables extend over concept or role names to capture the semantic properties of G-CORE. This leads to a number of required adaptations of the method presented in this work, since the knowledge base must encode the potential presence of previously unexpected concept or role names. The dataflow of the extension to [41] is indicated by red arrows in Figure 3. It is important to note that our mapping approach serves the formalization and implementation of schema inference; whether the queries are executed on a PG database or an RDF store is a separate question.

1.1 Contributions

In summary, the contributions of this paper are as follows:

1. We determine the impact of PG-specific expressiveness when it comes to shape inference for composable graph queries. Thus, we handle edges that have identities, as well as nodes and edges annotated with labels and key-value properties that persist, implicitly, in results.
2. Despite common DLs lacking the corresponding language constructs, we resolve the mismatch through reification, which carries over to structures of shapes and queries.
3. Instead of reification at the DL level, we define an intermediate layer, introducing mappings from ProGS to SHACL and G-CORE to SPARQL `CONSTRUCT` over PGs reified in RDF.
4. We develop the metatheory regarding the soundness of inferred shapes and the semantic equivalence of mapped shapes and queries.

1.2 Roadmap of the Paper

Section 2 introduces the underlying formalisms, as well as our running examples. In Section 3, we formalize the essential problem addressed in this work. Section 4 and Section 5 solve the problem in two steps: first, we develop a family of mappings from property graph abstractions to RDF graph abstractions (Section 4). On these grounds, we develop an inference method, which is an extension of [41] (Section 5). Section 6 develops the corresponding metatheory. In Section 7, we briefly outline our prototype implementation of this approach. Finally, Section 8 discusses related work while Section 9 concludes the paper.

In several places we refer to an extended version of the paper, referenced in the information section on the first page, which includes additional examples as well as full proofs.

2 Foundations

We first introduce the graph data models, query languages, and shape-based validation languages relevant to our work; they can be separated into two groups, namely *PG-related* languages and *RDF-related* languages pertaining to the upper and lower layers of Figure 3.

2.1 RDF Graphs

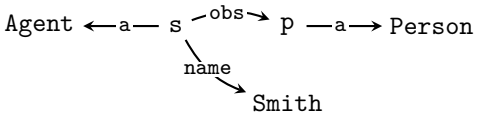
We define RDF graphs based on the W3C specification [13], excluding blank nodes and literals; we consider the latter as ordinary IRIs to simplify the formalization of the data model. We assume four pairwise disjoint, finite[1] but arbitrary, and sufficiently large sets of IRIs, namely concept

[1] This is not a restriction to the problem, since the sets are arbitrary, but it simplifies a few definitions.

names $\mathbf{C}$, individual names $\mathbf{I}$, role names $\mathbf{R}$, and the special role name {`rdf:type`}. For the sake of consistency, we use description-logic terminology throughout all abstractions (e.g., *concept name* instead of *class*). Commonly, RDF uses so-called prefixes to abbreviate (parts of) IRIs. In formal definitions, we express most IRIs through ordinary symbols, with a few exceptions, such as `rdf:type`. In examples, we also omit prefixes for the example domain; that is, we simply write `Agent` instead of `ex:Agent`. Similarly, we sometimes write `a` instead of `rdf:type`.

▶ **Definition 1** (Simple RDF Graph). *An* RDF graph G *is a finite set of triples of the form* $(a,\ p,\ b)$ *or* $(a,\ \texttt{rdf:type},\ A)$ *where* $a, b \in \mathbf{I}$*,* $A \in \mathbf{C}$*, and* $p \in \mathbf{R}$*.*

▶ **Example 2.** Consider the following set of triples as an example RDF graph $G_1^{rdf} = \{(\texttt{s}, \texttt{a}, \texttt{Agent}), (\texttt{s}, \texttt{name}, \texttt{Smith}), (\texttt{p}, \texttt{a}, \texttt{Person}), (\texttt{s}, \texttt{obs}, \texttt{p})\}$.



2.1.1 SPARQL Queries

We define a subset of SPARQL queries referred to as *Extended Conjunctive* `CONSTRUCT` *Queries* (ECCQ) in analogy to *Simple Conjunctive* `CONSTRUCT` *Queries* (as defined in [41]) in Definition 3. We choose this conjunctive fragment, as is commonly done, to keep the scope of our algorithms manageable; we extend[2] the fragment of SPARQL chosen in [41] with the language features required for our intended mappings.

To this end, we introduce *generic* patterns such as $x : y$, written `?x a ?y` in concrete SPARQL syntax, where both x and y are variables. Here, we need to introduce an additional syntactic restriction for generic variables, as we will discuss in more depth later: if the query includes the atomic query pattern $x : y$, then y must not occur again in another atomic query pattern. That is, we can copy all concept names of x generically but cannot introduce restrictions on this meta level; for example, a query with patterns $\{x : y, z : y\}$ is not allowed, since it would constrain the bindings of x and z through y. Thus, we write $x : \overline{x}^{\mathbf{C}}$ to indicate that the generic variable $\overline{x}^{\mathbf{C}}$ is syntactically dependent on x in this way. We also introduce certain filter expressions for only these generic variables written $\overline{x}^{\mathbf{C}}$, filtering certain concept or role names from their bindings.

▶ **Definition 3** (Syntax of ECCQ). *We define atomic patterns* t *and filter expressions* f *as follows:*

$$t ::= \ x : A \mid (x, y) : p \mid (x, a) : p \mid x : \overline{x}^{\mathbf{C}} \mid (x, \overline{x}^{\mathbf{I}}) : \overline{x}^{\mathbf{R}} \qquad f ::= \ \overline{x}^{\mathbf{C}} \neq A \mid \overline{x}^{\mathbf{R}} \neq p$$

with $A \in \mathbf{C}$*,* $a \in \mathbf{I}$*,* $p \in \mathbf{R}$*, and variables* $x, y, \overline{x}^{\mathbf{C}}, \overline{x}^{\mathbf{R}}, \overline{x}^{\mathbf{I}} \in \mathbf{V}$*. Let the function* vars *denote the set of all variables that occur in a set of atomic patterns or filter expressions. An ECCQ* q *is defined as* $H \leftarrow (P, F)$ *where* template H *and* pattern P *consist of finite sets of atomic patterns* t *and the* filters F *of a finite set of filter expressions* f *such that* $\text{vars}(F) \subseteq \text{vars}(P)$*. For each atomic pattern* $x : \overline{x}^{\mathbf{C}}$ *(and similarly* $(x, \overline{x}^{\mathbf{I}}) : \overline{x}^{\mathbf{R}}$*), there is a unique variable* $\overline{x}^{\mathbf{C}}$ *(similarly* $\overline{x}^{\mathbf{R}}$ *and* $\overline{x}^{\mathbf{I}}$*) per ordinary variable* x*, which does not occur again in the query, except in filter conditions. If a generic atomic pattern of this form occurs in* H*, the same pattern must also occur in* P*.*

[2] ECCQ is not a true extension of the language presented in [41], since we omit certain patterns, e.g., $a : C$. While they could be easily included, making ECCQ a true extension of this language, they are not required for the remainder of this work.

To formalize the semantics of ECCQ in Definition 4, we define valuations $\mu : \mathbf{V} \to \mathbf{I} \cup \mathbf{C} \cup \mathbf{R}$ as mappings from variables to individual, concept, and role names. Note that the syntactic restrictions on $\overline{x}^{\mathbf{C}}$ and $\overline{x}^{\mathbf{R}}$ ensure that the respective types of these bindings cannot be mixed up, since these variables can only occur in the correct context. Two valuations μ_1 and μ_2 are *compatible*, denoted $\mu_1 \sim \mu_2$, if for every variable x occurring in both μ_1 and μ_2, $\mu_1(x) = \mu_2(x)$. A finite set of valuations is denoted Ω. We define the *join* of two sets of valuations Ω_1 and Ω_2 as $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\}$.

Note that in Definition 3 we do not include the usual syntactic restriction $\mathrm{vars}(H) \subseteq \mathrm{vars}(P)$. We therefore need the final clause in Definition 4 about variables from atomic patterns $t \in H$ that do not occur in $\mathrm{vars}(P)$. Here, we assume the generation of fresh IRIs for such variables, which will be required by our mappings later. While this is a deviation from ordinary SPARQL semantics[3], our variant simplifies the formalization by avoiding the need to represent blank nodes in our RDF graph model and throughout the paper. In practical implementations, ordinary blank nodes should be used to comply with standard SPARQL semantics; indeed, our implementation generates SPARQL queries featuring blank node identifiers in their templates as well.

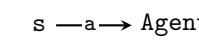
▶ **Definition 4** (Semantics of ECCQ). *The result of evaluating an ECCQ $H \leftarrow (P, F)$ over an RDF graph G is the RDF graph denoted $[\![H \leftarrow (P, F)]\!]_G^{eccq}$ and defined as*

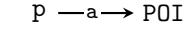
$$[\![H \leftarrow (P, F)]\!]_G^{eccq} = \{\mu(t) \mid \mu \in [\![P, F]\!]_G^{eccq\text{-}p}, t \in H\}$$

where $[\![P, F]\!]_G^{eccq\text{-}p} = \{\mu \mid \mu \in \bowtie_{t \in P} [\![t]\!]_G^{eccq\text{-}a}$, for all $f \in F : \mu \vdash f\}$ and $[\![t]\!]_G^{eccq\text{-}a} = \{\mu \mid \mu(t) \in G\}$. f is a filter condition such that $\mu \vdash (\overline{x}^{\mathbf{C}} \neq A)$ is true iff $\mu(\overline{x}^{\mathbf{C}})$ is not A, and $\mu \vdash (\overline{x}^{\mathbf{R}} \neq p)$ is true iff $\mu(\overline{x}^{\mathbf{R}})$ is not p. In a slight abuse of notation, we write $\mu(t)$ to mean $\mu(u : A) = (\mu(u),$ `rdf:type`, $A)$, and similarly for the remaining cases. If a variable x in $t \in H$ does not occur in $\mathrm{vars}(P)$*, then $\mu(x)$ produces a fresh IRI. This IRI is unique for each variable x and each unique mapping μ.*

▶ **Example 5.** Consider the ECCQ $q_1^{eccq} = \{\mathtt{x} : \mathtt{Agent}, \mathtt{y} : \mathtt{POI}\} \leftarrow (\{(\mathtt{x}, \mathtt{y}) : \mathtt{obs}\}, \{\})$, selecting pairs of observers and observed before constructing a new graph labeling observers as `Agent` and observed as `POI`. When evaluating this query on graph G_1^{rdf} from Example 2, written $[\![q_1^{eccq}]\!]_{G_1^{rdf}}$, we first match the pattern P on the graph, resulting in the single mapping μ where $\mu(\mathtt{x}) = s$ and $\mu(\mathtt{y}) = p$. Thus, the template H is instantiated only once, namely for μ, resulting in the graph $\{(\mathtt{s}, \mathtt{a}, \mathtt{Agent}), (\mathtt{p}, \mathtt{a}, \mathtt{POI})\}$.

Using generic patterns, we could, in this particular case, rewrite the query as $\{\mathtt{x} : \overline{\mathtt{x}}^{\mathbf{C}}, \mathtt{y} : \mathtt{POI}\} \leftarrow (\{\mathtt{x} : \overline{\mathtt{x}}^{\mathbf{C}}, (\mathtt{x}, \mathtt{y}) : \mathtt{obs}\}, \{\})$, copying all concepts for bindings for `x` instead of explicitly adding `Agent`. Since s has only the concept `Agent` in q_1^{eccq}, evaluation would yield the same result graph.





[3] Usually, SPARQL `CONSTRUCT` templates can feature blank node identifiers (`https://www.w3.org/TR/sparql12-query/#templatesWithBNodes`) to this effect.

2.1.2 SHACL Shapes

SHACL is a validation language for RDF graphs, where a shape consists of two components: the target query selecting a subset of nodes and a constraint these nodes must conform to. We present here the Seifer et al. [41] definition of SHACL shapes, utilizing description logics to define the semantics of target queries and constraints based on work by Bogaerts et al. [8].

According to this definition, SHACL shapes are $\mathcal{ALCHOI}$ axioms, i.e. target query subsumed by the constraint, both encoded as concept descriptions (Definition 6), and their semantics can be reduced to checking consistency with a *validation knowledge base* constructed for a given RDF graph G by its unique-name, closed-world, and domain-closure assumptions. We refer to [7] for the definition of the semantics of $\mathcal{ALCHOI}$. We give here the syntax of $\mathcal{ALCHOI}$ axioms, which can express a subset of SHACL, and the core idea of the validation semantics introduced in [41]; we refer to that paper for the full derivation and proofs.

▶ **Definition 6** ($\mathcal{ALCHOI}$ Axioms)**.** Concept descriptions *are defined as follows:*

$$C ::= \top \mid \bot \mid A \mid \{a\} \mid C \sqcap C \mid C \sqcup C \mid \neg C \mid \exists \rho.C \mid \forall \rho.C$$
$$\rho ::= p \mid p^{-}$$

where A, a, and p stand for concept names, individual names, and role names, respectively, and $\top$ and $\bot$ are two special concept names. Given two concept descriptions C and D, and two role descriptions ρ_1 and ρ_2, $C \sqsubseteq D$ and $\rho_1 \sqsubseteq \rho_2$ are $\mathcal{ALCHOI}$ axioms*, which we also call* shapes*. The left-hand side (C and ρ_1) we also call the* target *and the right-hand side (D and ρ_2) the* constraint *of the shape.*

Semantically, SHACL shapes target specific nodes of a graph, relying on constraints to validate these selected nodes. A consistency check for the subsumption axioms defined in Definition 6 with the validation knowledge base constructed for a given RDF graph G (Definition 7) encodes these semantics. Example 8 demonstrates the intuition.

▶ **Definition 7** (Validation Semantics (Adapted from [41]))**.** *The axioms of a simple RDF graph G, denoted $\mathcal{T}_G$, are the TBox consisting of the following $\mathcal{ALCHOI}$ axioms:*

1. Domain Closure Assumption (DCA)*: $\top \equiv \bigsqcup_{a \in \mathbf{I}} \{a\}$.*
2. Unique Name Assumption (UNA)*: $\{a\} \sqcap \{b\} \equiv \bot$, for each pair of distinct individual names $a, b \in \mathbf{I}$.*
3. Closed-World Assumption (CWA)*:*
   - *$A \equiv \bigsqcup_{a\,:\,A \in G} \{a\}$, for each concept name $A \in \mathbf{C}$,*
   - *$\exists p.\{a\} \equiv \bigsqcup_{(b,a)\,:\,p \in G} \{b\}$, and*
   - *$\exists p^{-}.\{a\} \equiv \bigsqcup_{(a,b)\,:\,p \in G} \{b\}$, for each role name $p \in \mathbf{R}$ and each individual name $a \in \mathbf{I}$.*

$(\mathcal{T}_G, G)$ is the validation knowledge base *of G. A graph G is* proof-valid *according to a set Σ of $\mathcal{ALCHOI}$ axioms if and only if Σ is consistent with the validation knowledge base of G, i.e. the knowledge base $(\mathcal{T}_G \cup \Sigma, G)$ admits a model.*

We write $\text{valid}(G, S)$ to indicate that G is proof-valid regarding the set of shapes S and $\text{valid}(G, s)$ to indicate that G is proof-valid regarding a single shape s. We use S and Σ interchangeably. In Example 8 we give an example shape and outline both the intuitive reason why this shape validates the running example graph and the formal reason considering the validation knowledge base.

▶ **Example 8.** Consider $s_1^{shacl} = \texttt{Agent} \sqsubseteq \exists \texttt{obs}.\texttt{Person}$ as an example. Intuitively, the shape expresses that all agents must observe at least one person. The graph G_1^{rdf} (Example 2) is valid regarding s_1^{shacl} since all targets, i.e. s, which is an instance of `Agent`, conform to the constraint, given both $(\texttt{s}, \texttt{obs}, \texttt{p}) \in G_1^{rdf}$ and $(\texttt{p}, \texttt{a}, \texttt{Person}) \in G_1^{rdf}$.

The validation knowledge base, constructed from the closed-world ($\texttt{Agent} \equiv \{\texttt{s}\}$, $\texttt{Person} \equiv \{\texttt{p}\}$, $\exists\texttt{obs}.\{p\} \equiv \{\texttt{s}\}$, and $\exists\texttt{obs}^{-}.\{s\} \equiv \{\texttt{p}\}$), domain-closure ($\{\texttt{s}\} \sqcup \{\texttt{p}\} \equiv \top$), and unique-name assumptions ($\{\texttt{s}\} \sqcap \{\texttt{p}\} \equiv \bot$) for G_1^{rdf} is consistent with this subsumption axiom, too: essentially, there is the concept $\{\texttt{s}\}$ which is equal to `Agent`, and also the range of `obs`; in turn, the domain of `obs` is $\{\texttt{p}\}$, which is also equal to `Person`.

2.2 Property Graphs

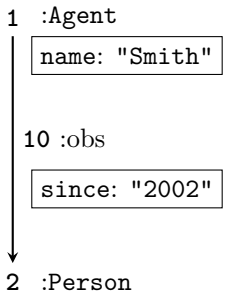
Property graphs (Definition 9) differ from RDF graphs in that edges have identities, and both nodes and edges can be annotated with sets of labels and sets of key-value pairs, the so-called properties. There are various notions of property graphs; we base our work largely on [3], that is, the graph model used for G-CORE, which is very similar to the graph model used in the context of Cypher[4] as well. We do not include first-class paths, however.

We define a set of labels $L = L_N \cup L_E$ where L_N is a finite set of node labels and L_E a finite set of edge labels; a finite set of property names (or keys) $K = K_N \cup K_E$ where K_N is a finite set of node keys and K_E a finite set of edge keys; and a finite set of literal values V. Without loss of generality, we assume that literal values are strings. Finally, let $\mathrm{FS}(s)$ denote all finite subsets of a set s, including the empty set. See Example 10 for the running example PG.

▶ **Definition 9** (Property Graph). *A* property graph *(PG) is a tuple $G = (N, E, \rho, \lambda, \sigma)$. N and E denote two disjoint, finite sets of node and edge identifiers, respectively. We write $n \in N$, $e \in E$, and $u \in N \cup E$. $\rho : E \to (N \times N)$ is a total function assigning edges to nodes; $\lambda : (N \cup E) \to FS(L)$ is a total function assigning labels to nodes or edges; and $\sigma : (N \cup E) \times K \to FS(V)$ is a total function assigning properties to nodes or edges and for which a set of tuples $(x, k) \in (N \cup E) \times K$ exists such that $\sigma(x, k) \neq \emptyset$.*

▶ **Example 10.** Consider $G_1^{pg} = (N_1, E_1, \rho_1, \lambda_1, \sigma_1)$, where $N_1 = \{\texttt{1}, \texttt{2}\}$ and $E_1 = \{\texttt{10}\}$ are the sets of node and edge IDs. A single edge is defined as $\rho_1(\texttt{10}) = (\texttt{1}, \texttt{2})$. Labels are defined as $\lambda_1(\texttt{1}) = \{\texttt{Agent}\}$, $\lambda_1(\texttt{2}) = \{\texttt{Person}\}$, and $\lambda_1(\texttt{10}) = \{\texttt{obs}\}$. Finally, we give the non-empty cases for the property function as $\sigma_1(\texttt{1}, \texttt{name}) = \{\texttt{"Smith"}\}$ and $\sigma_1(\texttt{10}, \texttt{since}) = \{\texttt{"2002"}\}$.

Compare this graph to the RDF graph G_1^{rdf} in Example 2: RDF graphs do not support properties on edges; thus, there is no direct equivalent for $\sigma_1(10, \texttt{since}) = \{\texttt{"2002"}\}$ in RDF. Otherwise, the graphs are equivalent.



2.2.1 Property Graph Queries

Composable property graph queries return graphs and function similarly to ECCQ queries. We define *Simple G-CORE Queries* (SGCQ) as a subset of G-CORE [3]. Our conjunctive fragment (Definition 11) supports matching multiple atomic patterns, including the most common constraints

[4] `https://opencypher.org/`

on labels and properties, and constructing a new graph from multiple atomic patterns while explicitly adding or removing labels and properties. In short, the fragment allows for selecting a subgraph, reshaping it, and adding constant elements. It does not support whole-graph operations, such as the union of a constructed graph with the original graph.

The basic semantics of SGCQ are similar to ECCQ: a *pattern* Γ defines variables and constrains their bindings through clauses ξ. These bindings are used to construct a new graph, specified in the *template* $\mathcal{B}$, together with *set* ($\oplus$) and *remove* ($\ominus$) clauses for explicitly adding or removing labels and properties. Unlike in the case of ECCQ, labels and properties persist in the result graph by default.

We give here in short the syntax and semantics of SGCQ adapted from [3], with the following change: since we allow only for a subset of filter expressions, we push these filters into the node (or edge) pattern. This is equivalent to the concrete syntax of G-CORE where we would write, e.g., `(x:Person)` to filter all nodes `x` for the label `Person`.

▶ **Definition 11** (Syntax of SGCQ). *A SGCQ $\mathcal{B} \Leftarrow \Gamma$ consists of a set Γ of atomic components γ (*pattern*) and a set $\mathcal{B}$ of atomic components β (*template*), defined by the following grammar:*

$$\gamma ::= (x_n, \xi) \mid (x_n, x_n){:}(x_e, \xi) \qquad \beta ::= (x_n, S) \mid (x_n, x_n){:}(x_e, S)$$

Here, x_n is a node variable, and x_e is an edge variable. ξ is a set of expressions of the form $:l$, $.k$, or $.k = v$, where $l \in L$ is a label, $k \in K$ is a property name, and $v \in V$ is a property value. S is a set of assignments for setting ($\oplus l$ and $\oplus k = v$) or removing ($\ominus l$ and $\ominus k$) labels or property names. We also write S_L for referring to operations involving labels and S_P for referring to operations involving properties. Furthermore, we write, e.g., $\ominus_x l \in \mathcal{B}$ to indicate that $(x, S) \in \mathcal{B}$ and $\ominus l \in S$. As an additional syntactical constraint of the template, edge variables that occur in the pattern may only occur when their associated node variables also occur in the template.

For the semantics of SGCQ, we again consider valuations μ on node or edge variables x_n or x_e, so that $\mu(x_n) \in N$ and $\mu(x_e) \in E$, and similar definitions for compatibility $\sim$ and the *join* operation $\bowtie$ as for *ECCQ*. We also define the union of two PGs $G_1 \cup G_2$ as $(N_1 \cup N_2, E_1 \cup E_2, \rho, \lambda, \sigma)$ where $\forall e \in E_1 \cup E_2 : \rho(e) = \rho_1(e)$ if $e \in E_1$ else $\rho_2(e)$, $\forall x \in N_1 \cup N_2 \cup E_1 \cup E_2, k \in K : \lambda(x) = \lambda_1(x) \cup \lambda_2(x)$, and $\sigma(x, k) = \sigma_1(x, k) \cup \sigma_2(x, k)$. The semantics for evaluating G-CORE queries is given in Definition 12, with the intuition outlined in Example 15.

▶ **Definition 12** (Semantics of SGCQ). *The result of evaluating a SGCQ $g = \mathcal{B} \Leftarrow \Gamma$ over a PG G is defined as the PG constructed by*

$$[\![q]\!]_G^{sgcq} = [\![\mathcal{B}]\!]_{\Omega,G}^{sgcq\text{-}t} \text{ where } \Omega = [\![\Gamma]\!]_G^{sgcq\text{-}p}.$$

That is, similarly to ECCQ queries, sets of bindings Ω are constructed from the pattern and fill out the template. $[\![\Gamma]\!]_G^{sgcq\text{-}p}$ and $[\![\mathcal{B}]\!]_{\Omega,G}^{sgcq\text{-}t}$ are specified in Definition 13 and Definition 14 based on [3], respectively.

▶ **Definition 13** (SGCQ Pattern Semantics). *The set of bindings Ω resulting from evaluation of a SGCQ pattern Γ over a graph G, written $[\![\Gamma]\!]_G^{sgcq\text{-}p}$, is defined by the following cases:*

$$\begin{aligned}
[\![(x,\xi)]\!]_G^{sgcq\text{-}p} &= \{\mu \mid \mu(x) \in N, \\
&\qquad \forall \xi_i \in \xi : [\![\xi_i]\!]_{\mu(x),G}^{sgcq\text{-}e} = \top\} \\
[\![(x,y){:}(z,\xi)]\!]_G^{sgcq\text{-}p} &= \{\mu \mid \mu(x), \mu(y) \in N, \mu(z) \in E, \\
&\qquad \rho(\mu(z)) = (\mu(x), \mu(y)), \\
&\qquad \forall \xi_i \in \xi : [\![\xi_i]\!]_{\mu(z),G}^{sgcq\text{-}e} = \top\} \\
[\![\Gamma]\!]_G^{sgcq\text{-}p} &= \bowtie_{\gamma \in \Gamma} [\![\gamma]\!]_G^{sgcq\text{-}p}
\end{aligned}$$

$$\begin{aligned}
[\![:l]\!]_{u,G}^{sgcq\text{-}e} &= \top \text{ iff } l \in \lambda(u) \\
[\![.k]\!]_{u,G}^{sgcq\text{-}e} &= \top \text{ iff } \sigma(u,k) \neq \emptyset \\
[\![.k = v]\!]_{u,G}^{sgcq\text{-}e} &= \top \text{ iff } v \in \sigma(u,k)
\end{aligned}$$

▶ **Definition 14** (SGCQ Template Semantics). *Evaluation of a SGCQ template $\mathcal{B}$ over a graph G, with bindings Ω, written $[\![\mathcal{B}]\!]^{sgcq\text{-}t}_{\Omega,G}$, is defined by the following cases:*

$$
\begin{aligned}
[\![(x,S)]\!]^{sgcq\text{-}t}_{\mu,G} &= \{\{v\},\emptyset,\emptyset,[\![S_L]\!]^{sgcq\text{-}s}_{v,G},[\![S_P]\!]^{sgcq\text{-}s}_{v,G}\}\\
[\![(x,y){:}(z,S)]\!]^{sgcq\text{-}t}_{\mu,G} &= \{\{v,u\},\{e\},\{e \rightarrow (v,u)\},[\![S_L]\!]^{sgcq\text{-}s}_{e,G},[\![S_P]\!]^{sgcq\text{-}s}_{e,G}\}\\
[\![\mathcal{B}]\!]^{sgcq\text{-}t}_{\Omega,G} &= \cup_{\beta\in\mathcal{B},\mu\in\Omega}[\![\beta]\!]^{sgcq\text{-}t}_{\mu,G}
\end{aligned}
$$

where $v=\mu(x)$, $u=\mu(y)$, and $e=\mu(z)$ if the variables are in the domain of μ, or fresh identities otherwise. Evaluation of set and remove clauses is defined as

$$
\begin{aligned}
[\![S_L]\!]^{sgcq\text{-}s}_{o_N,G} &= (\lambda_{o_N}\cup\{o_N \rightarrow l_N \mid \oplus l_N \in S_L\})\backslash\{o_N \rightarrow l_N \mid \ominus l_N \in S_L\}\\
[\![S_P]\!]^{sgcq\text{-}s}_{o_N,G} &= (\sigma'\cup\sigma_\oplus)\backslash\sigma_\ominus
\end{aligned}
$$

where

$$
\begin{aligned}
\sigma' &= \{(o_N,k_N)\rightarrow v \mid (o_N,k_N)\rightarrow v\in\sigma \wedge (o_N,k_N)\rightarrow v'\notin\sigma_\oplus\}\\
\sigma_\oplus &= \{(o_N,k_N)\rightarrow v \mid \oplus k_N = v\in S_P\}\\
\sigma_\ominus &= \{(o_N,k_N)\rightarrow v \mid (o_N,k_N)\rightarrow v\in\sigma \wedge \ominus k_N \in S_P\}
\end{aligned}
$$

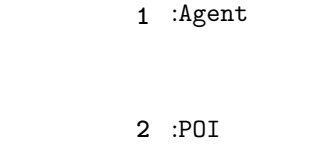
▶ **Example 15.** Recall the ECCQ q_1^{eccq} from Example 5. A *seemingly* similar SGCQ could be defined as

$$q_0^{sgcq} = \{(\mathtt{x},\{\oplus\mathtt{Agent}\}),(\mathtt{y},\{\oplus\mathtt{POI}\})\} \Leftarrow \{(\mathtt{x},\mathtt{y}){:}(\mathtt{e},\{\ {:}\mathtt{obs}\})\}$$

Evaluating $[\![q_0^{sgcq}]\!]^{sgcq}_{G_1^{pg}}$ (cf. Example 10) produces first the single mapping μ with $\mu(\mathtt{x})=\mathtt{1}$, $\mu(\mathtt{y})=\mathtt{2}$, and $\mu(\mathtt{e})=\mathtt{10}$; then, the template constructs the graph $G_0^{pg}=(\{\mathtt{1},\mathtt{2}\},\emptyset,\emptyset,\lambda_0,\sigma_0)$ with $\lambda_0(\mathtt{1})=\{\mathtt{Agent}\}$, $\lambda_0(\mathtt{2})=\{\mathtt{Person},\mathtt{POI}\}$, and $\sigma_0(\mathtt{1},\mathtt{name})=\{\mathtt{"Smith"}\}$. That is, all labels and properties persist in the output graph. The following update to q_0^{sgcq} has equivalent semantics to the ECCQ query:

$$q_1^{sgcq} = \{(\mathtt{x},\{\ominus\mathtt{name}\}),(\mathtt{y},\{\oplus\mathtt{POI},\ominus\mathtt{Person}\})\} \Leftarrow \{(\mathtt{x},\mathtt{y}){:}(\mathtt{e},\{\ {:}\mathtt{obs}\})\}$$

Evaluation produces the graph $(\{\mathtt{1},\mathtt{2}\},\emptyset,\emptyset,\lambda_1,\emptyset)$ with $\lambda_1(\mathtt{1})=\{\mathtt{Agent}\}$ and $\lambda_1(\mathtt{2})=\{\mathtt{POI}\}$, as we would expect (cf. Example 5).



2.2.2 Simple Property Graph Shapes (sProGS)

We define the syntax and semantics of a subset of the *Property Graphs Shapes Language* [44] (ProGS) we call *Simple Property Graph Shapes Language* (sProGS). The shape language is heavily inspired by SHACL and adapted for property graphs by differentiating between node and edge shapes, introducing constraints for property annotations, and for constraining the nodes (or edges, respectively) reachable from edges (or nodes, respectively).

We omit named shapes, simplifying shape semantics by avoiding recursive shapes (non-recursive named shapes are merely syntactic sugar), as well as some other features such as qualified number restrictions or complex path expressions. A sProGS shape is either a node shape ${}_N\langle q_N,\phi_N\rangle$ or an

$$[\![n]\!]_G^{pro\text{-}q} = \{n\} \qquad [\![e]\!]_G^{pro\text{-}q} = \{e\}$$
$$[\![l_N]\!]_G^{pro\text{-}q} = \{n \mid n \in N \wedge l_N \in \lambda(n)\} \qquad [\![l_E]\!]_G^{pro\text{-}q} = \{e \mid e \in E \wedge l_E \in \lambda(e)\}$$
$$[\![k_N]\!]_G^{pro\text{-}q} = \{n \mid n \in N \wedge \sigma(n, k_N) \neq \emptyset\} \qquad [\![k_E]\!]_G^{pro\text{-}q} = \{e \mid e \in E \wedge \sigma(e, k_E) \neq \emptyset\}$$

Figure 4 Evaluation of target node and target edge queries.

edge shape ${}_E\langle q_E, \phi_E\rangle$, where q_N and q_E are the *target queries*, and ϕ_N and ϕ_E are the respective *constraints* (see Definition 16 and Definition 17). Each node (or edge, respectively) of the graph that matches the target must satisfy the constraint.

▶ **Definition 16** (Node and Edge Targets). *Given target node or edge IDs n or e, node or edge labels l_N or l_E, and node or edge properties k_N or k_E, node or edge targets are defined as*

$$q_N ::= n \mid l_N \mid k_N$$
$$q_E ::= e \mid l_E \mid k_E$$

▶ **Definition 17** (Node and Edge Constraints). *Node and edge constraints are defined by*

$$\phi_N ::= \top \mid n \mid l_N \mid \neg\phi_N \mid \phi_N \wedge \phi_N \mid \exists k_N.(= v) \mid \exists k_N.\top \mid \exists^{\rightarrow}\phi_E \mid \exists^{\leftarrow}\phi_E$$
$$\phi_E ::= \top \mid e \mid l_E \mid \neg\phi_E \mid \phi_E \wedge \phi_E \mid \exists k_E.(= v) \mid \exists k_E.\top \mid \Rightarrow \phi_N \mid \Leftarrow \phi_N$$

where in particular $\exists k_N.(= v)$ and $\exists k_N.\top$ require properties with certain (v) or any ($\top$) values, $\exists^{\rightarrow}\phi_E$ requires an outgoing edge conforming to ϕ_E, and $\Rightarrow \phi_N$ requires the target node to conform to ϕ_N. The remaining cases are similar.

Definition 18 introduces the validation semantics for a PG under a set of sProGS shapes. Figure 4 defines the semantics of target node and target edge queries. Figure 5 defines the evaluation semantics of node and edge constraints. These definitions differ from [44], using direct evaluation of constraints on target nodes instead of a semantics of faithful assignments, which is not needed since we omit recursion.

▶ **Definition 18** (sProGS Validation). *A PG G is valid regarding a set of sProGS shapes S if it is valid regarding all shapes $s \in S$; validity regarding a single shape $s = {}_N\langle q_N, \phi_N\rangle$ (or $s = {}_E\langle q_E, \phi_E\rangle$, respectively), denoted* $\text{valid}(G, s)$*, is defined as follows:*

$$\text{valid}(G, {}_N\langle q_N, \phi_N\rangle) = \forall n \in [\![q_N]\!]_G^{pro\text{-}q} : [\![\phi_N]\!]_{n,G}^{pro\text{-}n} = \text{true}$$
$$\text{valid}(G, {}_E\langle q_E, \phi_E\rangle) = \forall e \in [\![q_E]\!]_G^{pro\text{-}q} : [\![\phi_E]\!]_{e,G}^{pro\text{-}e} = \text{true}$$

In a slight abuse of notation, we also write $\text{valid}(G, S)$ to indicate that G is valid regarding each $s \in S$. Example 19 shows two example shapes.

▶ **Example 19.** Recall s_1^{shacl} from Example 8. We can express an equivalent shape using sProGS with $s_1^{pro} = {}_N\langle \texttt{Agent}, \exists^{\rightarrow}(\texttt{obs} \wedge \Rightarrow \texttt{Person})\rangle$. This is a node shape that targets all nodes with the node label `Agent`; it requires for validity that these nodes have an outgoing edge ($\exists^{\rightarrow}\phi_E$) where the constraint ϕ_E – an edge constraint – must validate at least one edge. For this edge, we require the edge label `obs`, and then, expressed with $\Rightarrow \phi_N$, that the corresponding destination node conforms to ϕ_N, i.e., has the node label `Person`.

We can also express shapes that target edges, such as $s_2^{pro} = {}_E\langle \texttt{obs}, \Leftarrow \texttt{Agent} \wedge \Rightarrow \texttt{Person}\rangle$, which states that edges labeled with `obs` must have an origin node labeled `Agent` ($\Leftarrow$ `Agent`) and the destination node labeled `Person` ($\Rightarrow$ `Person`). The example graph G_1^{pg} introduced in Example 10 is valid regarding both s_1^{pro} and s_2^{pro}.

$$
\begin{aligned}
[\![\top]\!]^{pro\text{-}n}_{n,G} &= \mathit{true} \\
[\![n']\!]^{pro\text{-}n}_{n,G} &= \mathit{true} \text{ if } n' = n \\
[\![l_N]\!]^{pro\text{-}n}_{n,G} &= \mathit{true} \text{ if } \ l_N \in \lambda(n) \\
[\![\neg\phi_N]\!]^{pro\text{-}n}_{n,G} &= \mathit{true} \text{ if } \ \text{not } [\![\phi_N]\!]^{pro\text{-}n}_{n,G} \\
[\![\phi^1_N \wedge \phi^2_N]\!]^{pro\text{-}n}_{n,G} &= \mathit{true} \text{ if } [\![\phi^1_N]\!]^{pro\text{-}n}_{n,G} \text{ and } [\![\phi^2_N]\!]^{pro\text{-}n}_{n,G} \\
[\![\exists k.(= v)]\!]^{pro\text{-}n}_{n,G} &= \mathit{true} \text{ if } v \in \sigma(n,k) \\
[\![\exists k.\top]\!]^{pro\text{-}n}_{n,G} &= \mathit{true} \text{ if } \sigma(n,k) \neq \emptyset \\
[\![\exists^{\rightarrow}\phi_E]\!]^{pro\text{-}n}_{n,G} &= \mathit{true} \text{ if } \exists e \in E, n' \in N \text{ such that } \rho(e) = (n,n') \text{ and } [\![\phi_E]\!]^{pro\text{-}e}_{e,G} \\
[\![\exists^{\leftarrow}\phi_E]\!]^{pro\text{-}n}_{n,G} &= \mathit{true} \text{ if } \exists e \in E, n' \in N \text{ such that } \rho(e) = (n',n) \text{ and } [\![\phi_E]\!]^{pro\text{-}e}_{e,G} \\
[\![\Rightarrow \phi_N]\!]^{pro\text{-}e}_{e,G} &= [\![\phi_N]\!]^{pro\text{-}n}_{n_2,G} \text{ where } (n_1,n_2) = \rho(e) \\
[\![\Leftarrow \phi_N]\!]^{pro\text{-}e}_{e,G} &= [\![\phi_N]\!]^{pro\text{-}n}_{n_1,G} \text{ where } (n_1,n_2) = \rho(e)
\end{aligned}
$$

Figure 5 Evaluation of node and edge constraints over a property graph G, omitting some cases for edges that are analogous to the node variants.

3 Formalizing the Problem

The essential problem statement of this work is formalized in the signature of Algorithm 1: We aim to decide, given a sProGS shape s, whether all possible output graphs of a SGCQ query q are valid regarding this shape. The set of *possible output graphs* is defined as the set of graphs produced by q from any graph that is valid regarding a given set of sProGS shapes $\mathcal{S}_{\text{in}}$, which we call the *input shapes* of our problem. With this decision procedure, we can find the set of all shapes characterizing the possible output graphs by enumerating a finite set of candidates and testing each candidate.

We refer to our prior work [41] for a detailed discussion on when such a finite set exists or how to deal with cases where it does not exist. In short, for restricted classes of shapes, the set of relevant candidates that need to be considered can be shown to be finite (e.g., $\mathcal{ALCHOI}$-based shapes); for other cases, heuristics can be employed. Note how this allows our approach to work for compositions of queries: the output of a prior step becomes the input of the next[5].

Algorithm 1 High-level pseudocode outlining our core approach.

Input A finite set of shapes sProGS $\mathcal{S}_{\text{in}}$, a SGCQ $q = \mathcal{B} \Leftarrow \Gamma$, and a sProGS shape s.
Output Does valid($[\![q]\!]_{G_{\text{in}}}, s$) hold for every graph G_{in} where valid($G_{\text{in}}, \mathcal{S}_{\text{in}}$)?

1: $s' \leftarrow {}_{pro \mapsto shacl}(s)$
2: $\mathcal{S}'_{\text{in}} \leftarrow \{{}_{pro \mapsto shacl}(s_i) \mid s_i \in \mathcal{S}_{\text{in}}\}$
3: $q' \leftarrow {}_{sgcq \mapsto eccq}(q)$
4: $\Sigma \leftarrow \text{infer}(\mathcal{S}'_{\text{in}}, q')$
5: **return if** test(Σ, s') **then** YES **else** UNKNOWN

[5] Indeed, since, as we will later explore in detail, the input shapes are encoded as $\mathcal{ALCHOI}$ axioms, compositions of multiple queries do *not* require full exploration of the explicit set of output shapes but can rely on the inferred axioms directly.

As motivated in the introduction (Figure 3), we solve this problem by first mapping all components (that is, graphs, shapes, and queries) to an intermediate RDF layer and then utilizing an extended version of our algorithm presented in [41]. In Algorithm 1, we outline this approach in pseudocode: here, the functions ${}_{pro\mapsto shacl}(s)$ and ${}_{sgcq\mapsto eccq}(q)$ are the mapping functions for encoding sProGS shapes and SGCQ queries in SHACL and ECCQ, respectively. A final function for mapping the graphs themselves is not actually utilized – since our approach does not depend on concrete graph instances – but is required for the sake of formalizing the remaining mappings.

The functions *infer* and *test* refer to the extension of [41]: the function *infer* takes the RDF-mapped query and shapes, producing a set of DL axioms Σ; this step extends the algorithm from [41], accounting for the additional features included in ECCQ (cf. Definition 3). The function *test* takes these axioms and checks whether it can be concluded that s' is guaranteed to hold for all possible output graphs; this step utilizes entailment.

Based on this formal problem signature, we define Theorem 20. We will construct the missing components of Algorithm 1 in the following two sections (the family of mappings in Section 4 and the extended inference approach in Section 5). Finally, in Section 6, we will show that the theorem holds.

▶ **Theorem 20** (Soundness of Algorithm 1). *Given a finite set of shapes $\mathcal{S}_{\text{in}}$, a SGCQ $q = \mathcal{B} \Leftarrow \Gamma$, and a shape s, then* $\text{valid}([\![q]\!]_{G_{\text{in}}}, s)$ *holds for every graph G_{in} where* $\text{valid}(G_{\text{in}}, \mathcal{S}_{\text{in}})$ *if*

$$\text{test}(\text{infer}(\{{}_{pro\mapsto shacl}(s_i) \mid s_i \in \mathcal{S}_{\text{in}}\}, {}_{sgcq\mapsto eccq}(q)),\ {}_{pro\mapsto shacl}(s)).$$

4 A Family of Mappings on Reified Property Graphs

As outlined in Figure 3 and the previous section, we introduce a family of mappings: we map PGs G to RDF graphs ${}_{pg\mapsto rdf}(G)$, utilizing reification for first-class edges (Section 4.1); this mapping is not directly utilized by our approach but instead comprises the formal basis on which the remaining mappings are defined. We map sProGS shapes s to $\mathcal{ALCHOI}$-encoded SHACL shapes ${}_{pro\mapsto shacl}(s)$, accounting for the reification introduced with the graph mapping (Section 4.2). And finally, we map SGCQ q to ECCQ ${}_{sgcq\mapsto eccq}(q)$, accounting for both reification and various semantic differences between the languages (Section 4.3).

4.1 PG to RDF Mapping

We first assume a utility function $\widehat{\ }$ that maps the individual elements of PGs to unique IRIs. The details of this function are left abstract; however, in real-world implementations, such a function could, for example, utilize prefixes to differentiate between PG constructs, including node and edge IDs, labels, properties, and values. We also introduce `m:nte` (*node-to-edge*) and `m:etn` (*edge-to-node*) as role names, and `m:node` and `m:edge` as special concept names distinct from $\mathbf{C}$. The former are used for reification of first-class edges, the latter for differentiating nodes and edges explicitly[6].

The straightforward mapping ${}_{pg\mapsto rdf}(G)$ is specified in Definition 21 and demonstrated in Example 22. It retains all information and can be reversed by simply inverting the presented mapping function. It utilizes $\widehat{\ }$ to map all nodes and edges, including their associated labels and properties, to RDF triples; IRIs for both PG nodes and PG edges are derived from their IDs. Edges are reified using two triples and the special role names `m:nte` and `m:etn`.

[6] For the sake of readability, we assign concrete names, using `m` to hint at some unique *meta* prefix that could be used in practice; they can be arbitrary but unique names.

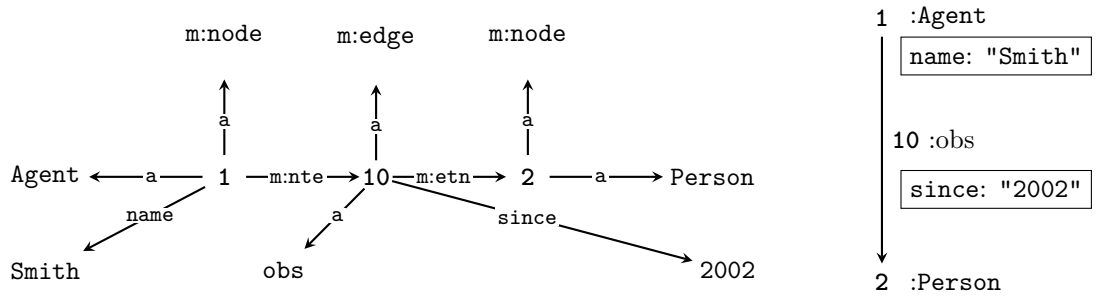


Figure 6 The reified RDF encoding ${}_{pg\mapsto rdf}(G_1^{pg})$ (left) and the original graph G_1^{pg} (right). Note how the edge with ID 10 (right) becomes a normal RDF node via reification (left).

▶ **Definition 21** (Mapping from PG to RDF Graph). *The corresponding RDF graph for a PG* $G = (N, E, \rho, \lambda, \sigma)$ *is the graph constructed by*

$$ {}_{pg\mapsto rdf}(G) \coloneqq \{{}^{node}_{pg\mapsto rdf}(n) \mid n \in N\} \cup \{{}^{edge}_{pg\mapsto rdf}(e) \mid e \in E\} $$

where ${}_{pg\mapsto rdf}$ *for nodes is defined as*

$$\begin{aligned} {}^{node}_{pg\mapsto rdf}(n) \coloneqq\ & \{(\widehat{n},\ \texttt{rdf:type},\ \widehat{l_N}) \mid l_N \in \lambda(n)\} \\ & \cup \{(\widehat{n},\ \widehat{k_N},\ \widehat{v}) \mid k_N \in K_N, v \in \sigma(n, k_N)\} \\ & \cup \{(\widehat{n},\ \texttt{rdf:type},\ \texttt{m:node})\} \end{aligned}$$

and for edges (with $\rho(e) = (n_1, n_2)$*) as*

$$\begin{aligned} {}^{edge}_{pg\mapsto rdf}(e) \coloneqq\ & \{(\widehat{n_1},\ \texttt{m:nte},\ \widehat{e}), (\widehat{e},\ \texttt{m:etn},\ \widehat{n_2})\} \\ & \cup \{(\widehat{e},\ \texttt{rdf:type},\ \widehat{l_E}) \mid l_E \in \lambda(e)\} \\ & \cup \{(\widehat{e},\ \widehat{k_E},\ \widehat{v}) \mid k_E \in K_E, v \in \sigma(e, k_E)\} \\ & \cup \{(\widehat{n},\ \texttt{rdf:type},\ \texttt{m:edge})\}. \end{aligned}$$

▶ **Example 22.** We map the graph G_1^{pg} from Example 10 to RDF, assuming a reasonable definition for $\widehat{\_}$ and omitting the display of any prefixes, to obtain the following set of triples for ${}_{pg\mapsto rdf}(G_1^{pg})$, as visualized side-by-side in Figure 6:

$$\begin{aligned} & \{(\texttt{1}, \texttt{a}, \texttt{m:node}), (\texttt{1}, \texttt{a}, \texttt{Agent}), (\texttt{1}, \texttt{name}, \texttt{Smith}), (\texttt{2}, \texttt{a}, \texttt{m:node}), (\texttt{2}, \texttt{a}, \texttt{Person}), \\ & (\texttt{10}, \texttt{a}, \texttt{m:edge}), (\texttt{10}, \texttt{a}, \texttt{obs}), (\texttt{10}, \texttt{since}, 2002), (\texttt{1}, \texttt{m:nte}, \texttt{10}), (\texttt{10}, \texttt{m:etn}, \texttt{2})\}. \end{aligned}$$

4.2 sProGS to SHACL ($\mathcal{ALCHOI}$) Mapping

We define a function ${}_{pro\mapsto shacl}$ for mapping sProGS shapes to SHACL shapes. Since we follow our prior work [41] in representing SHACL shapes as $\mathcal{ALCHOI}$ axioms, we map to $\mathcal{ALCHOI}$ directly. That is, we map both target queries and constraints to concept descriptions and shapes to subsumption axioms (Definition 23).

▶ **Definition 23** (Mapping sProGS to $\mathcal{ALCHOI}$). *A node shape* ${}_N\langle q_N, \phi_N\rangle$ *is mapped to the axiom* ${}_{pro\mapsto shacl}({}_N\langle q_N, \phi_N\rangle) = {}^{ntarget}_{pro\mapsto shacl}(q_N) \sqsubseteq {}^{nconstr}_{pro\mapsto shacl}(\phi_N)$*, where* ${}^{ntarget}_{pro\mapsto shacl}(q_N)$ *and* ${}^{nconstr}_{pro\mapsto shacl}(\phi_N)$ *are defined in Figure 7 and Figure 9, respectively. With equivalent definitions (Figure 8 and Figure 9), we can map edge shapes as* ${}_{pro\mapsto shacl}({}_E\langle q_E, \phi_E\rangle) = {}^{etarget}_{pro\mapsto shacl}(q_E) \sqsubseteq {}^{econstr}_{pro\mapsto shacl}(\phi_E)$.

sProGS	$\mathcal{ALCHOI}$
${}^{ntarget}_{pro\mapsto shacl}(n)$	$\{\widehat{n}\}$
${}^{ntarget}_{pro\mapsto shacl}(l_N)$	$\widehat{l_N}$
${}^{ntarget}_{pro\mapsto shacl}(k_N)$	$\exists\ \widehat{k_N}\ .\ \top$

Figure 7 Mapping for node target queries.

sProGS	$\mathcal{ALCHOI}$
${}^{etarget}_{pro\mapsto shacl}(e)$	$\{\widehat{e}\}$
${}^{etarget}_{pro\mapsto shacl}(l_E)$	$\widehat{l_E}$
${}^{etarget}_{pro\mapsto shacl}(k_E)$	$\exists\ \widehat{k_E}\ .\ \top$

Figure 8 Mapping for edge target queries.

sProGS	$\mathcal{ALCHOI}$
${}^{nconstr}_{pro\mapsto shacl}(\top)$	$\top$
${}^{nconstr}_{pro\mapsto shacl}(n)$	$\widehat{n}$
${}^{nconstr}_{pro\mapsto shacl}(l_N)$	$\widehat{l_N}$
${}^{nconstr}_{pro\mapsto shacl}(\neg\phi_N)$	$\neg {}^{nconstr}_{pro\mapsto shacl}(\phi_N)$
${}^{nconstr}_{pro\mapsto shacl}(\phi^1_N \wedge \phi^2_N)$	${}^{nconstr}_{pro\mapsto shacl}(\phi^1_N) \sqcap {}^{nconstr}_{pro\mapsto shacl}(\phi^2_N)$
${}^{nconstr}_{pro\mapsto shacl}(\exists k_N.(= v))$	$\exists\ \widehat{k_N}\ .\ v$
${}^{nconstr}_{pro\mapsto shacl}(\exists k_N.\top)$	$\exists\ \widehat{k_N}\ .\ \top$
${}^{nconstr}_{pro\mapsto shacl}(\exists^{\leftarrow}\phi_E)$	$\exists\ \texttt{m:etn}^-\ .\ {}^{econstr}_{pro\mapsto shacl}(\phi_E)$
${}^{nconstr}_{pro\mapsto shacl}(\exists^{\rightarrow}\phi_E)$	$\exists\ \texttt{m:nte}\ .\ {}^{econstr}_{pro\mapsto shacl}(\phi_E)$
...	...
${}^{econstr}_{pro\mapsto shacl}(\Rightarrow \phi_N)$	$\exists\ \texttt{m:etn}\ .\ {}^{nconstr}_{pro\mapsto shacl}(\phi_N)$
${}^{econstr}_{pro\mapsto shacl}(\Leftarrow \phi_N)$	$\exists\ \texttt{m:nte}^-\ .\ {}^{nconstr}_{pro\mapsto shacl}(\phi_N)$

Figure 9 Mapping sProGS node and edge constraints to $\mathcal{ALCHOI}$. We omit some cases for edge constraints that are equivalent to node constraint cases.

We write ${}_{pro\mapsto shacl}(S)$ as a shorthand for mapping each $s \in S$. Most constructs from the syntax of sProGS translate directly to $\mathcal{ALCHOI}$, similarly to how SHACL is defined in terms of $\mathcal{ALCHOI}$ in [41]. Interesting cases include the constraint $\Rightarrow \phi_N$, which maps to $\exists\ \texttt{m:etn}\ .\ {}^{nconstr}_{pro\mapsto shacl}(\phi_N)$. This node constraint requires the special role name `m:etn` according to the reification defined for PGs, since the original sProGS constraint refers to the target node of an edge. Given that this edge is an ordinary node in the RDF encoding, we must invoke $\exists$`m:etn` to navigate to the node representing that target and then use the mapped constraint ϕ_N. Similarly, $\exists^{\rightarrow}\phi_E$ is mapped to $\exists\ \texttt{m:nte}\ .\ {}^{econstr}_{pro\mapsto shacl}(\phi_E)$. Example 24 shows the full mappings for the running example.

▶ **Example 24.** We map the shapes s_1^{pro} and s_2^{pro} from Example 19 to equivalent $\mathcal{ALCHOI}$ subsumption axioms. For the node shape s_1^{pro}, this results in the following shape:

$$\begin{aligned} {}_{pro\mapsto shacl}(s_1^{pro}) &= {}_{pro\mapsto shacl}({}_N\langle \texttt{Agent}, \exists^{\rightarrow}(\texttt{obs} \wedge\ \Rightarrow \texttt{Person})\rangle) \\ &= \texttt{Agent} \sqsubseteq \exists \texttt{m:nte}.(\texttt{obs} \sqcap \exists \texttt{m:etn}.\texttt{Person}) \end{aligned}$$

Note how the reification is expressed as two nested existential quantifications: we first navigate to the reified edge (*node*) via `m:nte`, where not only a label (`obs`) but also a particular node constraint holds, to which we navigate via `m:etn`; we ensure `1:Agent` $\rightarrow_{\texttt{m:nte}}$ `10:obs` $\rightarrow_{\texttt{m:etn}}$ `2:Person` (cf. Figure 6). Similarly, for the edge shape s_2^{pro} we want `1:Agent` $\leftarrow_{\texttt{m:nte}}$ `10:obs` $\rightarrow_{\texttt{m:etn}}$ `2:Person`, but from the perspective of all reified edges, instead of nodes, such that in

$$\begin{aligned} {}_{pro\mapsto shacl}(s_2^{pro}) &= {}_{pro\mapsto shacl}({}_E\langle \texttt{obs},\ \Leftarrow \texttt{Agent} \wedge\ \Rightarrow \texttt{Person}\rangle) \\ &= \texttt{obs} \sqsubseteq \exists \texttt{m:nte}^-.\texttt{Agent} \sqcap \exists \texttt{m:etn}.\texttt{Person} \end{aligned}$$

we target an edge to navigate back to the origin (inverse `m:nte`) as well as the destination via `m:etn`, requiring the node labels `Agent` and `Person`, respectively.

4.3 SGCQ to ECCQ Mapping

Finally, we map SGCQ to ECCQ. The principal challenge for this mapping – aside from accounting for reification – is that in SGCQ the semantics for constructing labels and properties are different from ECCQ in that all properties and labels are kept for constructed nodes or edges unless there are remove clauses. In ECCQ, on the other hand, all triples in the result graph must be explicitly constructed by atomic patterns in the query template.

We solve this issue by utilizing generic concept and role variables with appropriate filter conditions to define a mapping ${}_{sgcq\mapsto eccq}(q)$ for a SGCQ q (Definition 25). To this end, we introduce an intermediate representation referred to as IQL (*Intermediate Query Language*) in Definition 26 that aligns with the two types of patterns of SGCQ, i.e. matching nodes or edges, on the SGCQ side with full query patterns or templates on the ECCQ side. This aligns parts of queries on both sides that compose naturally. Note how the decision to syntactically include where clauses in node or edge patterns of SGCQ aids this composability.

Via this intermediate layer, we associate SGCQ variables with generic copy patterns expressed in ECCQ that explicitly construct the required labels and properties. *Set* clauses translate to explicit construction in ECCQ, while *remove* clauses translate to filter conditions on generic variables in the pattern of ECCQ queries, removing the offending labels from the bindings for these generic variables. We demonstrate this mapping in Example 27.

▶ **Definition 25** (Mapping *SGCQ* to *ECCQ*). *Given a SGCQ q, we define the mapping function ${}_{sgcq\mapsto eccq}(q)$ by introducing the intermediate query language IQL (Definition 26) and several utility functions for mapping to and from IQL, as*

$$ {}_{sgcq\mapsto eccq}(q) := {}_{iql\mapsto eccq}({}_{sgcq\mapsto iql}(q)) $$

where the following utility functions (defined in Figure 10) are utilized:

$$ {}_{sgcq\mapsto iql}(\mathcal{B} \Leftarrow \Gamma) = \bigcup_{\beta\in\mathcal{B}} {}^{template}_{sgcq\mapsto iql}(\beta, \mathrm{vars}(\Gamma)) \cup \bigcup_{\gamma\in\Gamma} {}^{pattern}_{sgcq\mapsto iql}(\gamma) $$

$$ {}_{iql\mapsto eccq}(I_T \leftarrow I_P) = \bigcup_{t\in I_T} {}^{template}_{iql\mapsto eccq}(t) \cup \bigcup_{p\in I_P} {}^{pattern}_{iql\mapsto eccq}(p) $$

▶ **Definition 26** (Intermediate Query Language). *We define the syntax of the intermediate query language (IQL) as follows:*

$$ q := I_T \leftarrow I_P $$
$$ p := \mathrm{M_N}(x_n, W_L, W_K, W_V, R_L, R_K) \mid \mathrm{M_E}(x_n, x_n, x_e, W_L, W_K, W_V, R_L, R_K) $$
$$ t := \mathrm{C_N}(x_n, A_L, A_K, V) \mid \mathrm{C_E}(x_n, x_n, x_e, A_L, A_K, V) $$

where W_L is a set of label patterns $:l$, W_K is a set of key patterns $.k$, and W_V is a set of patterns $.k = v$. Similarly, A_L is a set of set label *patterns $\oplus l$, A_K is a set of* set key-value *patterns $\oplus k = v$, and V is a set of variables. Finally, R_L is a set of* remove label *patterns $\ominus l$, and R_K is a set of* remove key *patterns $\ominus k$. The semantics of IQL are implicitly defined through the mappings in Figure 10.*

▶ **Example 27.** The example SGCQ q_2^{sgcq} in Figure 11 (left) includes the major SGCQ features, such as matching on an edge pattern, setting and removing labels, and restricting results with constraints. The resulting ECCQ ${}_{sgcq\mapsto eccq}(q_2^{sgcq})$ is shown in Figure 11 (right). The query is suitable for application to G_1^{pg} from Example 10. We focus here on the single variable `y`, which occurs in two atomic patterns: (`y`, `:Person`) (in Γ) and (`y`, $\{\oplus$`POI`, $\ominus$`Person`$\}$) (in $\mathcal{B}$).

$\mathrm{M_N}$(`y`, {: `Person`}, $\emptyset, \emptyset,$ {$\ominus$`Person`}, $\emptyset$}) and $\mathrm{C_N}$(`y`, {$\oplus$`POI`}, $\emptyset, \emptyset,$ {`x`, `y`, `e`}) are the corresponding IQL terms. The former maps to the ECCQ pattern including the generic pattern $\mathtt{y} : \overline{\mathtt{y}}^{\mathbf{C}}$, copying all labels for `y`, the explicit pattern `y:Person` selecting instances of `Person`, but also the filter $\overline{\mathtt{y}}^{\mathbf{C}} \neq$ `Person`, which prevents bindings, and thus also the construction, for label `Person` for the generic variable $\overline{\mathtt{y}}^{\mathbf{C}}$, included because of the remove term in the original SGCQ query. The ECCQ template includes, along with the generic patterns, the explicit construction of `y:POI`.

$$
\begin{aligned}
{}^{pattern}_{sgcq\mapsto iql}((x,\xi)) :=& \mathrm{M_N}(x,\{\ :l \mid\ \ :l \in \xi\},\{\ .k \mid\ \ .k \in \xi\},\\
&\{\ .k = v \mid\ \ .k = v \in \xi\},\{\ominus l \mid \ominus l \in S_x\},\{\ominus k \mid \ominus k \in S_x\})\\
{}^{pattern}_{sgcq\mapsto iql}(((x,y){:}z,\xi)) :=& \mathrm{M_E}(x,y,z,\{\ :l \mid\ \ :l \in \xi\},\{\ .k \mid\ \ .k \in \xi\},\\
&\{\ .k = v \mid\ \ .k = v \in \xi\}\{\ominus l \mid \ominus l \in S_z\},\{\ominus k \mid \ominus k \in S_z\})
\end{aligned}
$$

$$
\begin{aligned}
{}^{template}_{sgcq\mapsto iql}((x,S),\mathcal{V}) :=& \mathrm{C_N}(x,\{\oplus l \mid \oplus l \in S\},\{\oplus k = v \mid \oplus k = v \in S\},V)\\
{}^{template}_{sgcq\mapsto iql}(((x,y){:}z,S),V) :=& \mathrm{C_E}(x,y,z,\{\oplus l \mid \oplus l \in S\},\{\oplus k = v \mid \oplus k = v \in S\},V)
\end{aligned}
$$

$$
\begin{aligned}
{}^{pattern}_{iql\mapsto eccq}(\mathrm{M_N}(\ldots)) :=& (\{x : \overline{x}^{\mathbf{C}}, (x,\overline{x}^{\mathbf{I}}) : \overline{x}^{\mathbf{R}}, x : \texttt{m:node}\} \cup\ \{x : \widehat{l} \mid\ \ :l \in W_L\}\\
&\cup\ \{(x,o_{x,k}) : \widehat{k} \mid\ \ .k \in W_K\} \cup\ \{(x,\widehat{v}) : \widehat{k} \mid\ \ .k = v \in W_V\},\\
&\{\overline{x}^{\mathbf{R}} \neq \texttt{m:nte}\} \cup \{\overline{x}^{\mathbf{C}} \neq \widehat{l} \mid \ominus l \in R_L\} \cup \{\overline{x}^{\mathbf{R}} \neq \widehat{k} \mid \ominus k \in R_K\})\\
{}^{pattern}_{iql\mapsto eccq}(\mathrm{M_E}(\ldots)) :=& (\{(x,z) : \texttt{m:nte}, (z,y) : \texttt{m:etn}, z : \overline{z}^{\mathbf{C}}, (z,\overline{z}^{\mathbf{I}}) : \overline{z}^{\mathbf{R}}, z : \texttt{m:edge}\}\\
&\cup\ \{z : \widehat{l} \mid\ \ :l \in W_L\} \cup\ \{(z,o_{z,k}) : \widehat{k} \mid\ \ .k \in W_K\}\\
&\cup\ \{(z,\widehat{v}) : \widehat{k} \mid\ \ .k = v \in W_V\},\\
&\{\overline{z}^{\mathbf{R}} \neq \texttt{m:etn}\} \cup \{\overline{z}^{\mathbf{C}} \neq \widehat{l} \mid \ominus l \in R_L\} \cup \{\overline{z}^{\mathbf{R}} \neq \widehat{k} \mid \ominus k \in R_K\})
\end{aligned}
$$

$$
\begin{aligned}
{}^{template}_{iql\mapsto eccq}(\mathrm{C_N}(\ldots)) :=& \{x : \texttt{m:node}\} \cup (\{x : \overline{x}^{\mathbf{C}}, (x,\overline{x}^{\mathbf{I}}) : \overline{x}^{\mathbf{R}}\} \text{ if } x \in V \text{ else } \{\})\\
&\cup\ \{x : \widehat{l} \mid \oplus l \in A_L\} \cup\ \{(x,\widehat{v}) : \widehat{k} \mid \oplus k = v \in A_K\}\\
{}^{template}_{iql\mapsto eccq}(\mathrm{C_E}(\ldots)) :=& \{(x,z) : \texttt{m:nte}, (z,y) : \texttt{m:etn}, z : \texttt{m:edge}\}\\
&\cup (\{z : \overline{z}^{\mathbf{C}}, (z,\overline{z}^{\mathbf{I}}) : \overline{z}^{\mathbf{R}}\} \text{ if } z \in V \text{ else } \{\})\\
&\cup\ \{x : \widehat{l} \mid \oplus l \in A_L\} \cup\ \{(x,\widehat{v}) : \widehat{k} \mid \oplus k = v \in A_K\}
\end{aligned}
$$

Figure 10 Component-wise mappings from SGCQ to IQL and IQL to ECCQ, both for query templates and patterns. We write S_x to refer to the union of all sets of add or remove patterns S, such that $(x,S) \in \mathcal{B}$ or $(x_1,x_2){:}(x,S) \in \mathcal{B}$.

5 Shape Inference for ECCQ

As introduced in the formalization of our core problem in Section 3, we aim to infer a set of $\mathcal{ALCHOI}$ axioms Σ from an ECCQ q and a set of shapes $\mathcal{S}_{\text{in}}$ (written $\Sigma = \text{infer}(\mathcal{S}_{\text{in}}, q)$), such that $\Sigma \models s_o$ ($\text{test}(\Sigma, s_o)$ in Section 3) implies that s_o is a shape validating all possible output graphs of q, under the precondition that the input graphs of q are guaranteed to be valid regarding $\mathcal{S}_{\text{in}}$.

5.1 Encoding Constraints in DL

We build on and extend [41], which essentially approximates axioms of the closed-world assumption from sub-expressions of the query and joins them with the input shapes and a few additional axioms. To this end, three namespaces need to be distinguished: those of the input graphs, of the subgraphs matched by the query, and of the new output graphs constructed as a result. The extensions of concepts involved clearly differ between these three namespaces; thus, we follow [41] in renaming any concept name A in the input namespaces as $\dot{A}$ and $\ddot{A}$ in the matched and output namespaces. We do the same for each role name p (with $\dot{p}$ and $\ddot{p}$).

$$
\begin{array}{l}
\{(\mathtt{x}, \emptyset), \\
\ (\mathtt{x}, \mathtt{y}){:}(\mathtt{e}, \emptyset), \\
\ (\mathtt{y}, \{\oplus\mathtt{POI}, \ominus\mathtt{Person}\})\} \\
\ \Leftarrow \\
\{(\mathtt{x}, \{\ .\mathtt{name} = \texttt{"Smith"}\}), \\
\ (\mathtt{x}, \mathtt{y}){:}(\mathtt{e}, \{\ :\mathtt{obs}\}), \\
\ (\mathtt{y}, \{\ :\mathtt{Person}\})\}
\end{array}
\qquad
\begin{array}{l}
\{\mathtt{x}:\mathtt{m{:}node}, \mathtt{x}:\overline{\mathtt{x}}^{\mathbf{C}}, (\mathtt{x}, \overline{\mathtt{x}}^{\mathbf{I}}):\overline{\mathtt{x}}^{\mathbf{R}}, \\
\ \mathtt{e}:\mathtt{m{:}edge}, \mathtt{e}:\overline{\mathtt{e}}^{\mathbf{C}}, (\mathtt{e}, \overline{\mathtt{e}}^{\mathbf{I}}):\overline{\mathtt{e}}^{\mathbf{R}}, (\mathtt{x}, \mathtt{e}):\mathtt{m{:}nte}, (\mathtt{e}, \mathtt{y}):\mathtt{m{:}etn}, \\
\ \mathtt{y}:\mathtt{m{:}node}, \mathtt{y}:\overline{\mathtt{y}}^{\mathbf{C}}, (\mathtt{y}, \overline{\mathtt{y}}^{\mathbf{I}}):\overline{\mathtt{y}}^{\mathbf{R}}, \mathtt{y}:\mathtt{POI}\} \\
\ \leftarrow \\
(\{\mathtt{x}:\mathtt{m{:}node}, \mathtt{x}:\overline{\mathtt{x}}^{\mathbf{C}}, (\mathtt{x}, \overline{\mathtt{x}}^{\mathbf{I}}):\overline{\mathtt{x}}^{\mathbf{R}}, (\mathtt{x}, \mathtt{Smith}):\mathtt{name}, \\
\ \mathtt{e}:\mathtt{m{:}edge}, \mathtt{e}:\overline{\mathtt{e}}^{\mathbf{C}}, (\mathtt{e}, \overline{\mathtt{e}}^{\mathbf{I}}):\overline{\mathtt{e}}^{\mathbf{R}}, \mathtt{e}:\mathtt{obs}, (x, \mathtt{e}):\mathtt{m{:}nte}, (\mathtt{e}, \mathtt{y}):\mathtt{m{:}etn}, \\
\ \mathtt{y}:\mathtt{m{:}node}, \mathtt{y}:\overline{\mathtt{y}}^{\mathbf{C}}, (\mathtt{y}, \overline{\mathtt{y}}^{\mathbf{I}}):\overline{\mathtt{y}}^{\mathbf{R}}, \mathtt{y}:\mathtt{Person}\}, \\
\ \{\overline{\mathtt{y}}^{\mathbf{C}} \neq \mathtt{Person}, \overline{\mathtt{x}}^{\mathbf{R}} \neq \mathtt{m{:}nte}, \overline{\mathtt{e}}^{\mathbf{R}} \neq \mathtt{m{:}etn}, \overline{\mathtt{y}}^{\mathbf{R}} \neq \mathtt{m{:}nte}\})
\end{array}
$$

Figure 11 SGCQ q_2^{sgcq} on the left, and ECCQ ${}_{sgcq \mapsto eccq}(q_2^{sgcq})$ on the right. Note how we visually align the lines of the left and right columns for their common variables, hinting at the IQL rules, e.g., the `POI` set label and respective triple pattern (blue). The only exception is the inclusion of $\overline{\mathtt{y}}^{\mathbf{C}} \neq \mathtt{Person}$ (yellow) in the filter patterns of the ECCQ (right column), highlighting why the IQL pattern M_N needs the remove clauses as part of their arguments: remove clauses from the template of SGCQ affect the pattern (filters) of ECCQ.

(a) An example input graph. **(b)** Variable concepts. **(c)** Output graph of q. **(d)** Output graph of q'.

Figure 12 Example for graphs involved in constructing the validation knowledge base (cf. [41]) for the query q with $P = H = \{x:\mathtt{A}, x:\mathtt{B}, y:\mathtt{E}\}$ from Example 28. This case demonstrates the potential issue arising when adding the atomic pattern $y:\overline{y}^{\mathbf{C}}$ in q', since variable y matches `b` which is an instance of `A`; this violates the assumptions about the relationship of $\ddot{\mathtt{A}}$ and $\ddot{\mathtt{B}}$ in the output graph. Colors visualize different namespaces, while floating labels indicate concept names (e.g., a is both `A` and `B`).

Query variables can be represented with *variable concepts*, that is, fresh concept names V_x for each variable x. The extensions of these concepts are equal to the set of all of their bindings. Thus, variable concepts remain the same in all namespaces and are not subject to any renaming. They are constrained by the basic graph patterns of the query pattern P they occur in and therefore play a key role in encoding the input-output relationship of queries.

The essential extension of our SPARQL subset ECCQ over the subset considered in [41] are generic copy operations through variables for concepts and properties and certain filter expressions on these variables. The minimal Example 28 shows how this invalidates the axioms inferred by [41], while also giving the intuition of this method.

▶ **Example 28.** Assume a query q with $P = H = \{\mathtt{x}:\mathtt{A}, \mathtt{x}:\mathtt{B}, \mathtt{y}:\mathtt{E}\}$ and no input shapes. The axioms inferred by [41] would include $\{\mathtt{V_x} \equiv \mathtt{A} \sqcap \mathtt{B}, \mathtt{V_y} \equiv \mathtt{E}, \ddot{\mathtt{A}} \equiv \mathtt{V_x}, \ddot{\mathtt{B}} \equiv \mathtt{V_x}, \ddot{\mathtt{E}} \equiv \mathtt{V_y}\}$. We demonstrate these for an example input graph (Figure 12a) in Figure 12. That is, we define the variable concept $\mathtt{V_x}$ (visualized as a graph in Figure 12b) in terms of its bindings, namely the intersection of `A` and `B`, and the new concept names in the output namespace $\ddot{\mathtt{A}}$ and $\ddot{\mathtt{B}}$ in terms of these $\mathtt{V_x}$. Since these axioms entail, e.g., $\ddot{\mathtt{A}} \sqsubseteq \ddot{\mathtt{B}}$, just as in the example Figure 12c, we can conclude that the shape $\mathtt{A} \sqsubseteq \mathtt{B}$ holds on all output graphs.

If we were to extend query q as q' by adding $\{\mathtt{y} : \overline{\mathtt{y}}^{\mathbf{C}}\}$ to both P and H, the axiom $\ddot{\mathtt{A}} \sqsubseteq \ddot{\mathtt{B}}$ *should* no longer be entailed; indeed, variable $\mathtt{y}$ might now also include – such as in Figure 12d – bindings that are of type $\mathtt{A}$ exclusive or $\mathtt{B}$ and modify $\ddot{\mathtt{A}}$ or $\ddot{\mathtt{B}}$ through the generic pattern $\mathtt{y} : \overline{\mathtt{y}}^{\mathbf{C}}$, thereby invalidating this subsumption.

We can remedy this broken inference introduced in Example 28 by including *components* of all variables that have generic copy operations in the definition of concepts like $\ddot{\mathtt{A}}$. Such components are defined in Definition 30 and included in axioms constructed by the adapted method outlined in Definition 31. We write $\mathrm{vcg}(q)$ (*variable connectivity graph*) to mean a graph where variables $\mathrm{var}(q)$ are nodes and there exists an edge between two variables if and only if they both occur in a pattern (e.g., x and y in $(x, y) : p$) in ${}_{sgcq \mapsto eccq}(q)$. Example 28 can therefore be fixed by using this extension as shown in Example 29. We demonstrate a full example of the algorithm in Section 5.3.

▶ **Example 29.** With the extended method, we infer the set of axioms including $\{\mathtt{V_x} \equiv \mathtt{A} \sqcap \mathtt{B}, \mathtt{V_y} \equiv E, \ddot{\mathtt{A}} \equiv \mathtt{V_x} \sqcup \mathtt{V}^{\mathtt{A}}_{\mathtt{y}} \sqcup \mathtt{V}^{\mathtt{A}}_{\mathtt{x}}, \ddot{\mathtt{B}} \equiv \mathtt{V_x} \sqcup \mathtt{V}^{\mathtt{B}}_{\mathtt{x}} \sqcup \mathtt{V}^{\mathtt{B}}_{\mathtt{y}}, \ddot{E} \equiv \mathtt{V_y} \sqcup \mathtt{V}^{E}_{\mathtt{x}} \sqcup \mathtt{V}^{E}_{\mathtt{y}}\}$ where $\mathtt{V}^{\mathtt{A}}_{\mathtt{y}} \equiv \mathtt{V_y} \sqcap \mathtt{A}$, $\mathtt{V}^{\mathtt{B}}_{\mathtt{y}} \equiv \mathtt{V_y} \sqcap \mathtt{B}$, $\mathtt{V}^{\mathtt{E}}_{\mathtt{y}} \equiv \mathtt{V_y} \sqcap E$ and $\mathtt{V}^{\mathtt{A}}_{\mathtt{x}} \equiv \mathtt{V}^{\mathtt{B}}_{\mathtt{x}} \equiv \mathtt{V}^{E}_{\mathtt{x}} \equiv \bot$.

Here, the axioms defined for the output namespace (e.g., $\ddot{\mathtt{A}} \equiv \mathtt{V_x} \sqcup \mathtt{V}^{\mathtt{A}}_{\mathtt{x}} \sqcup \mathtt{V}^{\mathtt{A}}_{\mathtt{y}}$) include additional component concepts (e.g., $\mathtt{V}^{\mathtt{A}}_{\mathtt{y}}$) for all variables in the query. Thus, we ensure that the possibility of atomic pattern $\mathtt{y} : \overline{\mathtt{y}}^{\mathbf{C}}$ constructing instances of $\ddot{\mathtt{A}}$ in output graphs is accounted for. As a result, we can no longer *erroneously* infer $\mathtt{A} \sqsubseteq \mathtt{B}$ without additional knowledge in terms of input shapes about the relationship of $\mathtt{A}$, $\mathtt{B}$, and $\mathtt{E}$.

▶ **Definition 30** (Concept Components). *We define the* concept components $\mathrm{Cc}(q)$ *of an ECCQ* $q = H \leftarrow (P, F)$ *as the set of axioms including:*

1. *For all* $x \in \mathrm{var}(P)$ *and* $A \in \mathcal{V}$*, if* $x : \overline{x}^{\mathbf{C}} \in P \cap H$ *and* $(\overline{x}^{\mathbf{C}} \neq A) \notin F$*, then* $V_x^A \equiv V_x \sqcap A$*. Otherwise,* $V_x^A \equiv \bot$*.*
2. *For all* $x, y \in \mathrm{var}(P)$ *and* $p \in \mathcal{V}$*, if* $(x, \overline{x}^{\mathbf{I}}) : \overline{x}^{\mathbf{R}} \in P \cap H$*,* $(\overline{x}^{\mathbf{R}} \neq p) \notin F$ *and* $(x, y) : p \notin P$*, then* $V_x^p \equiv V_x \sqcap \exists p. V_x^{p,o}$*. Otherwise,* $V_x^p \equiv \bot$ *and* $V_x^{p,o} \equiv \bot$*.*

Here, $\mathcal{V} = \mathrm{voc}(q) \cup \mathrm{voc}(\mathcal{S}_{\mathrm{in}})$ *is the* vocabulary *of all concept, role, and individual names that occur in the query or in input shapes.*

▶ **Definition 31** (Axiom Encoding). *The* axiom encoding *of an ECCQ* q *and a set of* $\mathcal{ALCHOI}$ *shapes* $\mathcal{S}_{\mathrm{in}}$*, denoted* $\mathrm{infer}(\mathcal{S}_{\mathrm{in}}, q)$*, is the minimal set of* $\mathcal{ALCHOI}$ *axioms that include* $\mathrm{Cc}(q)$ *(Definition 30) as well as the following axioms:*

1. *The set of input shapes* $\mathcal{S}_{\mathrm{in}}$*.*
2. *For each variable* x *in* $\mathrm{var}(q)$*, the axiom*

$$V_x \sqsubseteq \bigsqcap_{x\,:\,A \in P} A \sqcap \bigsqcap_{(x,u)\,:\,p \in P} \exists p.\, \mathrm{C}_u \sqcap \bigsqcap_{(u,x)\,:\,p \in P} \exists p^-.\, \mathrm{C}_u,$$

and if $\mathrm{vcg}(P \cup H)$ *is acyclic regarding* x*, then also the axiom*

$$V_x \sqsupseteq \bigsqcap_{x\,:\,A \in P} A \sqcap \bigsqcap_{(x,u)\,:\,p \in P} \exists p.\, \mathrm{C}_u \sqcap \bigsqcap_{(u,x)\,:\,p \in P} \exists p^-.\, \mathrm{C}_u\,.$$

3. *For each concept name* A *in* $\mathrm{voc}(H)$*,* $\ddot{A} \equiv \bigsqcup_{u\,:\,A \in H} \mathrm{C}_u \sqcup \bigsqcup_{x \in \mathrm{var}(H)} V_x^A$*.*

4. *For each* p *in* $\mathrm{voc}(H)$

$$\begin{aligned}
\textstyle\bigsqcup_{(u,v)\,:\,p\in H} \mathrm{C}_u &\sqsubseteq \exists \ddot{p}.\, \mathrm{C}_v,\\
\exists \ddot{p}.\, \mathrm{C}_v &\sqsubseteq \textstyle\bigsqcup_{(u,v)\,:\,p\in H} \mathrm{C}_u \sqcup \bigsqcup_{(x,\overline{x}^{\mathbf{I}})\,:\,\overline{x}^{\mathbf{R}}\in H} V_x,\\
\exists \ddot{p}.\, \top &\equiv \textstyle\bigsqcup_{(u,v)\,:\,p\in H} (\mathrm{C}_u \sqcap \exists \ddot{p}.\, \mathrm{C}_v) \sqcup \bigsqcup_{x\in \mathrm{var}(H)} V_x^p,\\
\textstyle\bigsqcup_{(u,v)\,:\,p\in H} \mathrm{C}_v &\sqsubseteq \exists \ddot{p}^{-}.\, \mathrm{C}_u,\\
\exists \ddot{p}^{-}.\, \mathrm{C}_u &\sqsubseteq \textstyle\bigsqcup_{(u,v)\,:\,p\in H} \mathrm{C}_v \sqcup \bigsqcup_{(x,\overline{x}^{\mathbf{I}})\,:\,\overline{x}^{\mathbf{R}}\in H} V_x^{p,o},\\
\exists \ddot{p}^{-}.\, \top &\equiv \textstyle\bigsqcup_{(u,v)\,:\,p\in P} (\mathrm{C}_v \sqcap \exists \ddot{p}^{-}.\, \mathrm{C}_u).
\end{aligned}$$

5. *For each* $p \in P$, $\dot{p} \sqsubseteq p$.
6. *For each* $p \in P$, $p \sqsubseteq \dot{p}$ *if all atomic patterns including* p *in* P *have the form* $(x, y) : p$ *where* x, y *occur in no other atomic pattern in* P *and* $x \neq y$.
7. *For each pair* p, r *with* $(x, y) : p \in P$ *and* $(x, y) : r \in H$ *or* $(y, x) : r \in H$
   a. $\dot{p} \sqsubseteq \ddot{r}$ *(if* $(x, y) : r \in H$*) or* $\dot{p} \sqsubseteq \ddot{r}^{-}$ *(if* $(y, x) : r \in H$*) – if* P *does not contain any other atomic patterns with* p*, and*
   b. $\ddot{r} \sqsubseteq \dot{p}$ *(if* $(x, y) : r \in H$*) or* $\ddot{r}^{-} \sqsubseteq \dot{p}$ *(if* $(y, x) : r \in H$*) – if* H *does not contain any other atomic patterns with* r.

Cases 1, 2, and 5-7 are taken from, and 3 and 4 are adapted from [41].

5.2 Enriching the Query

Generic copy operations can invalidate output shapes in certain cases, as shown in Example 29. On the other hand, they enable means of semantically enriching the ECCQ query, leading to additional shapes that *can* be shown to hold. To this end, we define the extension rules in Definition 32 that essentially copy constraints from the query pattern to the template in cases where this is allowed by generic patterns. We include an example in Section 5.3.

▶ **Definition 32** (Extension of ECCQ). *The semantics-preserving extension of a ECCQ* $q = H \leftarrow (P, F)$*, written* $\mathrm{Ext}(q)$*, is defined by inserting the following atomic patterns to the template* H*:*

1. *If* $x : \overline{x}^{\mathbf{C}} \in P \cap H$*, then for each* $x : A \in P$ *such that* $(\overline{x}^{\mathbf{C}} \neq A) \notin F$*, we include the atomic pattern* $x : A$ *in* H*, and*
2. *if* $(x, \overline{x}^{\mathbf{I}}) : \overline{x}^{\mathbf{R}} \in P \cap H$*, then for each* $(x, a) : p \in P$ *such that* $(\overline{x}^{\mathbf{R}} \neq p) \notin F$*, we include the atomic pattern* $(x, a) : p$ *in* H.

5.3 Example

We now demonstrate some axioms in Σ constructed for the problem inputs $S_1^{pro} = \{s_1^{pro}, s_2^{pro}\}$ (Example 19) and q_2^{sgcq} (Example 27). Again, we will focus on axioms related to concept names, as they are more intuitive, yet in principle very similar to axioms related to role names. As a first step, we need to map the query to ECCQ as shown in Example 27 and then extend the mapped query to obtain the ECCQ input as $q_2^{eccq} = \mathrm{Ext}({}_{sgcq\mapsto eccq}(q_2^{sgcq}))$. The extended query template includes the atomic patterns $\{\mathtt{e} : \mathtt{obs}, (\mathtt{x}, \mathtt{"Smith"}) : \mathtt{name}\}$ in addition to the atomic patterns that are shown in Figure 11, which are copied from the query pattern due to the presence of generic patterns for `x` and `e`. On the other hand, since the only constraining atomic pattern for `y`, namely $\mathtt{y} : \mathtt{Person}$, is also included in the filter $\overline{\mathtt{y}}^{\mathbf{C}} \neq \mathtt{Person}$, there is no template extension for variable `y`, even though there are generic patterns (cf. Definition 32).

According to Definition 31.1 we need to include the input shapes in Σ after mapping them to $\mathcal{ALCHOI}$ axioms via ${}_{pro\mapsto shacl}(S_1^{pro})$. These axioms encode constraints that are known to hold on the input graph by definition, since we only consider valid input graphs (cf. Example 24).

According to Definition 31.2 we next define variable concepts encoding known constraints for the bindings of all query variables. We include the following axioms in Σ:

$$\mathtt{V_e} \equiv \exists \mathtt{m{:}etn}.\mathtt{V_y} \sqcap \exists \mathtt{m{:}nte^{-}}.\mathtt{V_x} \sqcap \mathtt{obs}$$
$$\mathtt{V_x} \equiv \exists \mathtt{name}.\mathtt{Smith} \sqcap \exists \mathtt{m{:}nte}.\mathtt{V_e}$$
$$\mathtt{V_y} \equiv \exists \mathtt{m{:}etn^{-}}.\mathtt{V_e} \sqcap \mathtt{Person}$$

These variable concepts are almost direct translations of the query pattern; for example, variable $\mathtt{x}$ occurs in $(\mathtt{x}, \mathtt{Smith}) : \mathtt{name}$ and $(\mathtt{x}, \mathtt{e}) : \mathtt{m{:}nte}$ (Example 27), which translates to the intersection of $\exists \mathtt{name}.\mathtt{Smith}$ and $\exists \mathtt{m{:}nte}.\mathtt{V_e}$ in the definition of $\mathtt{V_x}$. Note how reification again plays a role here, with the presence of the role name $\mathtt{m{:}nte}$.

Constraints related to concept names that explicitly occur in the query template are defined according to Definition 31.3 in terms of these variable concepts, including the concept components that were introduced in Definition 30. In the following set, we also include one axiom inferred from the rules for properties, namely $\mathtt{V_e} \sqsubseteq \exists \mathtt{m{:}\dot{e}tn}.\mathtt{V_y}$.

$$\mathtt{o\dot{b}s} \equiv \mathtt{V_e} \sqcup \mathtt{V_e^{obs}}$$
$$\mathtt{Ag\dot{e}nt} \equiv \mathtt{V_x^{Agent}} \sqcup \mathtt{V_y^{Agent}}$$
$$\mathtt{V_x^{Agent}} \equiv \mathtt{V_x} \sqcap \mathtt{Agent}$$
$$\mathtt{V_y^{Agent}} \equiv \mathtt{V_y} \sqcap \mathtt{Agent}$$
$$\mathtt{V_e^{obs}} \equiv \mathtt{V_e}$$
$$\mathtt{P\ddot{O}I} \equiv \mathtt{V_y} \sqcup \mathtt{V_x^{POI}} \sqcup \mathtt{V_y^{POI}}$$
$$\mathtt{V_x^{POI}} \equiv \mathtt{V_x} \sqcap \mathtt{POI}$$
$$\mathtt{V_y^{POI}} \equiv \mathtt{V_y} \sqcap \mathtt{POI}$$
$$\mathtt{V_e} \sqsubseteq \exists \mathtt{m{:}\dot{e}tn}.\mathtt{V_y}$$

There are a few noteworthy aspects to these axioms: firstly, note how $\mathtt{P\ddot{O}I}$ (defined with dots, since it differs from the concept $\mathtt{POI}$, which may occur in input graphs) is defined in terms of $\mathtt{V_y}$, because it exclusively occurs in the atomic pattern $\mathtt{y} : \mathtt{POI}$ of the query template, and $\mathtt{V_y}$ encodes the bindings of variable $\mathtt{y}$. Furthermore, we must include the concept components $\mathtt{V_x^{POI}}$ (and also $\mathtt{V_y^{POI}}$, though this is redundant in this case) in the definition of $\mathtt{P\ddot{O}I}$ since variable $\mathtt{x}$ occurs with the generic pattern $\mathtt{x} : \bar{\mathtt{x}}^{\mathbf{C}}$, meaning that the new concept $\mathtt{P\ddot{O}I}$ may extend over some bindings for $\mathtt{x}$, too. We treat $\mathtt{Ag\dot{e}nt}$ similarly, with the difference that there is no actual construct pattern featuring $\mathtt{Agent}$; thus, its definition consists only of concept components.

The complete set of inferred axioms entails, e.g., $\mathtt{o\dot{b}s} \sqsubseteq \exists \mathtt{m{:}\dot{e}tn}.\mathtt{P\ddot{O}I}$. Therefore, we know that $\mathtt{obs} \sqsubseteq \exists \mathtt{m{:}etn}.\mathtt{POI}$ validates all output graphs of the query. For the final shapes, we drop the namespace distinction, since it is not part of our (or any regular) query semantics.

6 Metatheory

To encode our inference task in $\mathcal{ALCHOI}$, we first map shapes and queries to $\mathcal{ALCHOI}$ and ECCQ then infer a set of axioms Σ from these mapped queries and shapes (cf. Algorithm 1). Therefore, we need to prove that any mapped shape $C \sqsubseteq D$ holds on all output graphs of the query if $\Sigma \models \dot{C} \sqsubseteq \dot{D}$. That is, if our inferred set of axioms Σ entails the respective axiom (i.e. encoded shape) in the output graph namespace annotated with two dots.

The primary outcome of the following considerations is the soundness of Algorithm 1, as expressed in Theorem 20. To this end, we first prove the semantic equivalence for our mappings in Proposition 33 and Proposition 34. Full proofs are available in our extended version.

▶ **Proposition 33** (Soundness of Shape Mapping). *Given a PG G and a set of sProGS shapes S, the graph G is valid regarding each shape $s \in S$ if and only if the mapped graph ${}_{pg \mapsto rdf}(G)$ is valid regarding the mapped shapes ${}_{pro \mapsto shacl}(s)$.*

Proof Sketch. We prove the proposition in two steps. First, we show that given a sProGS shape s, the targets of s in G correspond to the set of targets of ${}_{pro\mapsto shacl}(s)$ in ${}_{pg\mapsto rdf}(G)$. Then, we show that for any node (or edge) in G, it conforms to the constraint of a shape s if and only if the mapped node in ${}_{pg\mapsto rdf}(G)$ conforms to the constraint of ${}_{pro\mapsto shacl}(s)$. ◀

▶ **Proposition 34** (Soundness of Query Mapping). *Given a PG G and a SGCQ q, the following holds:* $[\![q]\!]^{sgcq}_G = {}_{pg\mapsto rdf}{}^{-1}([\![{}_{sgcq\mapsto eccq}(q)]\!]^{eccq}_{{}_{pg\mapsto rdf}(G)})$*, where* ${}_{pg\mapsto rdf}{}^{-1}$ *is the inverse mapping of* ${}_{pg\mapsto rdf}$*. That is, evaluating the query q is the same as evaluating the mapped query on the mapped graph and then mapping the result graph back.*

Proof Sketch. We separate the *matching* and *constructing* parts of queries for this proof. To this end, we rely on the composable intermediate query language (IQL), making individual node or edge patterns compose neatly with full ECCQ patterns (or templates). We define an equivalence relation between sets of bindings Ω for queries expressed as both SGCQ and ECCQ, which is not straightforward, given that SGCQ queries bind only nodes and edges (while labels and properties are implicit), but the corresponding reified labels and properties in ECCQ are explicitly bound as variables. We prove that both SGCQ and mapped ECCQ produce equivalent bindings. In the second part of the proof, we show that – given sets of equivalent bindings – the constructed graphs are again the same, modulo the mapping relation. ◀

It remains to show that shapes entailed by the axioms Σ constructed according to Definition 31 are guaranteed to validate output graphs of the query, modulo renaming in the appropriate namespace.

▶ **Proposition 35.** *Given an ECCQ q and a set of $\mathcal{ALCHOI}$ shapes $\mathcal{S}_{\text{in}}$ it holds for all RDF graphs G with* $\text{valid}(G, \mathcal{S}_{\text{in}})$ *that* $\text{infer}(\mathcal{S}_{\text{in}}, q) \models \ddot{s}$ *implies* $\text{valid}([\![q]\!]^{eccq}_G, s)$*.*

Proof Sketch. For this proof we first formalize the different conceptual graphs involved (according to the namespaces separated with one or two dots) as joined *extended graphs* and then show for the axioms constructed by the rules of Definition 31 that all extended graphs are indeed valid regarding these axioms. From this, we then follow (utilizing some auxiliary lemmas) that any other entailed shapes, when renamed accordingly, validate all possible output graphs of the query. This proof builds on the proofs in [41]. ◀

Finally, we show that extending the ECCQ preserves query semantics.

▶ **Proposition 36.** *For any ECCQ q and RDF graph G,* $[\![q]\!]^{eccq}_G = [\![\text{Ext}(q)]\!]^{eccq}_G$*.*

Proof Sketch. This proposition follows directly from the specified semantics of ECCQ since all concept or role names explicitly added to the template of q by rules for $\text{Ext}(q)$ (Definition 32) are also constructed by the respective generic patterns $x : \overline{x}^{\mathbf{C}}$ and $(x, \overline{x}^{\mathbf{I}}) : \overline{x}^{\mathbf{R}}$, except those that occur in filter conditions, which are excluded in the definition of $\text{Ext}(q)$. ◀

With these propositions, we can finally address the main claim of this paper. We rewrite the theorem initially introduced as Theorem 20 in Theorem 37, replacing the function *test* with concrete entailment and including the family of mappings. The proof follows directly from the propositions discussed in this section.

▶ **Theorem 37** (Reformulation of Theorem 20). *Given a finite set of sProGS shapes $\mathcal{S}_{\text{in}}$, a SGCQ $q = \mathcal{B} \Leftarrow \Gamma$, and a sProGS shape s, then* $\text{valid}([\![q]\!]_{G_{\text{in}}}, s)$ *holds for every graph G_{in} where* $\text{valid}(G_{\text{in}}, \mathcal{S}_{\text{in}})$ *if* $\text{infer}({}_{pro\mapsto shacl}(\mathcal{S}_{\text{in}}), {}_{sgcq\mapsto eccq}(q)) \vdash {}_{pro\mapsto shacl}(\ddot{s})$*.*

Proof Sketch. This follows directly from the propositions introduced in this section. ◀

7 Implementation and Evaluation

We implemented the algorithm described in this paper, including both the mapping and inference components, in Scala. To this end, we built on our previous implementation of [41] (see [42]), which is available under a free software license. Our fork of this implementation is available on GitHub[7] [43] as well.

We implemented parsers, as well as our formally specified mappings, for both G-CORE to SPARQL `CONSTRUCT` and ProGS to SHACL; modified the inference algorithm of the implementation [42] with our extended variant as specified, constructing different sets of description logic axioms that are then passed to the HermiT [28] reasoner for inference; implemented unit and end-to-end test cases that cover G-CORE and the extended SPARQL features; and validated the algorithm based on query execution over randomly generated graphs.

7.1 Experimental Setup and Results

To validate the correctness of our implementation and gather empirical evidence, in addition to the proofs provided by this work, on the correctness of our method, we generated random problem instances, consisting of a set of input shapes and queries. We generated random queries first and then sampled shapes based on their vocabularies, extended with additional random components that do not occur in queries. For each sample input, we applied our algorithm to obtain a set of output shapes; to this end, we generated a set of candidate shapes and checked each for entailment. We only kept samples that produced at least one valid output shape. We covered four different scenarios labelled corresponding to the nature of the shapes involved. While *small* and *large* mainly refer to the number of shapes, the *wide* sample includes conjunction and disjunction, whereas the *deep* sample includes more deeply nested shapes.

Figure 13 shows the distribution of a few key properties across the experiments we performed, including the basic structure of queries and shapes, as well as the candidate shapes that were considered. For additional details, we refer to our extended version or the documentation of the implementation itself. The upper half of the tables describes queries, while the lower half corresponds to shapes. For both *wide* and *large* we report only data on shapes, since the queries are mostly identical to the *deep* scenario. Regarding queries, we indicate the number of basic node or edge atomic clauses in the pattern and template, the number of set, remove, or where clauses, and the distinct number of labels and keys. Regarding shapes, we include the number of sampled input shapes, their structure (number of basic components and clauses in conjunction or disjunction), the number of candidates over the vocabulary tested as potential output shapes, and the inferred set of output shapes, i.e., the subset of candidates that is entailed by the constructed knowledge base. Our scenarios were designed to cover various interactions of query structures and shapes and facilitate our validation approach (random sample graphs), not necessarily to correspond to real-world data, where a baseline is also not available.

Our validation framework operates fully on the mapped SHACL and SPARQL level, taking the input SHACL shapes mapped from ProGS as the basis for generating a random RDF graph valid regarding these shapes. The generation approach is rather brute force, to not bias the sample graphs with additional information from shapes or queries, taking only information from the vocabulary of the problem instance while removing any parts that violate input shapes. We next applied the (mapped-from G-CORE) SPARQL `CONSTRUCT` query to this sample graph to obtain the corresponding output graph. Finally, we validated the output graph with the inferred

[7] `https://github.com/softlang/s2s`

Type	Avg.	Med.
MATCH	1.00	1.00
CONSTRUCT	1.00	1.00
WHEN Clauses	3.15	3.00
SET Clauses	1.59	2.00
REMOVE Clauses	0.99	1.00
Distinct Labels	3.67	4.00
Distinct Keys	1.31	1.00
Input Shapes	1.00	1.00
Components	2.16	2.00
Clauses	1.00	1.00
Candidates	256.24	240.00
Output Shapes	7.54	2.00

(a) Sample *small* (25,000).

Type	Avg.	Med.
MATCH	1.21	1.00
CONSTRUCT	2.21	2.00
WHEN Clauses	3.20	3.00
SET Clauses	1.57	2.00
REMOVE Clauses	0.99	1.00
Distinct Labels	3.64	4.00
Distinct Keys	1.33	1.00
Input Shapes	2.63	3.00
Components	2.96	3.00
Clauses	1.00	1.00
Candidates	408.25	370.00
Output Shapes	7.53	2.00

(b) Sample *deep* (25,000).

Type	Avg.	Med.
Input Shapes	2.61	3.00
Components	4.95	5.00
Clauses	1.86	2.00
Candidates	510.25	468.00
Output Shapes	9.85	2.00

(c) Sample *wide* (25,000).

Type	Avg.	Med.
Input Shapes	5.07	5.00
Components	2.16	2.00
Clauses	1.00	1.00
Candidates	610.32	576.00
Output Shapes	11.18	2.00

(d) Sample *large* (25,000).

Figure 13 Overview of statistical measures across different generator configurations.

output shapes. For each sample, we report either the class of a generation-related issue (leading to vacuous satisfaction of output shapes), output graph validation failure (shape violations), or success (non-vacuously valid graphs).

In $100,000$ samples we observed *no* validation failures, providing evidence towards the correctness of our method and implementation. A total of $37,959$ samples were full successes, meaning that the output graph was not only valid regarding the output shapes but also included all their targets. In $12,858$ cases, the output graph only included a subset of the targets present in output shapes, while in 266 cases, the input graph included only a subset of all input shape targets; we consider these partial successes. The remainder ($48,917$ samples) were other structural issues, where in most cases result graphs were either entirely empty ($26,591$) or did not include any of the required target nodes ($15,560$), while in fewer cases we were unable to produce a suitable input graph ($6,622$), or the query exceeded the allowed execution time of 60 seconds (144). These results did not vary significantly between the different classes.

Note that in all of these cases, modulo the few timeouts where we do not know the result, the output shapes were still vacuously satisfied. Since our validation attempts each sample generation up to 100 times, we hypothesize that the reason for these cases mostly lies in the structure of the sample itself: in these cases, graphs conforming to input shapes are unlikely to be matched by the query, since both do not align well. Again, we decided against optimizing for these scenarios to avoid overfitting generated samples to *preferred* cases, which could bias results.

We validated the implementation of the validation itself with 1,000 samples over the same four classes, where output shapes (7 per sample) were drawn randomly from the set of candidate shapes instead of utilizing the inference procedure; here, we obtain 76.7% validation failures and only 0.2% complete successes, indicating a very high probability for detecting failures. Our complete results, including additional metadata for each sample, results for the individual classes of samples, all details regarding the structure and generation of samples, and the tools to replicate these and similar experiments, are provided with our implementation.

7.2 Runtime Performance

While the original project, on which our implementation is based, includes tooling to demonstrate feasibility in terms of runtime performance (cf. [41]), the results would be difficult to compare between the different classes of queries and shapes used in our implementation. Additionally, we lack datasets for realistic properties of G-CORE or ProGS queries and shapes. Instead, we therefore report only a preliminary indication of the feasibility of runtime performance, within the same order of magnitude as the prior results, by measuring the inference time (testing all shape candidates reported in Figure 13) for all 100,000 samples generated for the correctness validation above. Here, we observe average (median) inference times per sample of 13.94ms (11.00ms) for *small*, 34.81ms (25.00ms) for *deep*, 42.23ms (29.50ms) for *wide*, and 51.68ms (36.00ms) for *large* samples on commodity hardware (Ryzen 7 9700X), without any dedicated Scala optimization.

8 Related Work

Mappings between query languages and related (graph) data models have been considered for various languages (cf. [31]), including mappings among graph query languages (e.g., [48]) as well as between graph query languages and SQL (e.g., [39, 12, 46]). Closely related to our mapping problem, SPARQL `SELECT` queries have been mapped to Cypher [52, 2]. However, in this direction, the divide between RDF and PG is less challenging, given that PG can directly represent RDF graphs. Reification of PG as RDF (e.g., [33, 51]) has been discussed under consideration of querying performance [24], though only under manually defined, ad hoc reformulation of Cypher queries in SPARQL `SELECT`.

At the data model level, the lack of interoperability between RDF and PG is a known issue [5], and mappings from RDF to PG have been considered as well, e.g., [6]. In [37], the authors consider both RDF graphs and SHACL shapes, transforming them to PG and PG-Schema, respectively. RDF-star [19] is an extension to RDF for the representation of metadata as nested triples, allowing for similar features as PG, thus working towards bridging the formal gap between these data models. Indeed, both PG mappings to RDF-star [20, 24], and the reverse [1] have been defined. In particular, this also includes execution of SPARQL-star queries over PG [20].

In contrast, in our work, we leverage reification in pure RDF for the sake of utilizing DL via SHACL, and we are concerned with bridging the semantic differences between graph construction in G-CORE, with its implicit labels and properties, and SPARQL `CONSTRUCT`, with its explicit triple construction, which are not addressed in previous work.

Inference of shapes or schemas can be approached essentially in two ways:

Option 1 – inference from instance data, e.g., concrete graphs or tables;

Option 2 – inference from queries and other functions constructing data.

The first option has been considered for RDF using various algorithms based on logical, statistical, or ML-driven inference for constructing SHACL or ShEx [36] shapes from sample graph instances [45, 16, 38, 32, 34, 17, 9]. Similar approaches have been considered recently for PG and their schema languages [29, 11, 17].

In our work, we follow the second option and do not consider any instance data but instead infer shapes valid in the context of any possible graph that queries might operate on in the future. However, inference from concrete graphs may complement our approach when initial input shapes for queries should be determined. Our approach leverages a prior method [41], as discussed throughout the paper, but we cover PG and G-CORE through mappings; in due course, we also extend the subset of SPARQL considered in [41] with generic copy patterns.

Related work on the inference of SHACL shapes from direct mappings [49] and RML rules [14] also follows the second option, but these approaches lack the arbitrary construction through query patterns we support, in addition to explicit input constraints. Views over relational databases can be considered equivalent mechanisms to composable graph queries; SQL schemas have been inferred for such views in [25, 15, 47, 22] though suffer from infeasibility of first-order formulas (as compared to DL) or restricted semantics of functional or join dependencies, which are orthogonal to constructs supported by ProGS in our case. [10] solves a similar problem, focusing on basic cardinality constraints (as graph schemas) and formally defined graph transformations (conjunctive path queries) over a concrete query language. They, too, rely on description logics as an encoding mechanism.

9 Conclusion

In this paper, we have presented an algorithm for inferring ProGS shapes validating all possible property graphs constructed by queries specified in a subset of G-CORE. Our algorithm encodes constraints inferred from the underlying query, as well as any known shapes that hold on the inputs of the query. Internally, we rely on the description logic $\mathcal{ALCHOI}$ by constructing a knowledge base that entails a sound set of shapes validating all output graphs of this query.

Since this DL encoding requires a reification step for first-class edges with labels and properties, we choose a clean, intermediate abstraction layer between property graphs and $\mathcal{ALCHOI}$, namely the SPARQL `CONSTRUCT` query language and SHACL shapes, over property graphs reified in RDF. This also allows us to reuse an underlying algorithm defined for these languages, which we must extend to support a larger subset of SPARQL, including the features required by our mapping. We prove the soundness of this shape inference approach and the semantic equivalence of the shapes and queries constructed by our mappings.

As future work, a larger, non-conjunctive subset of G-CORE could be supported by extending both the mapping and the capabilities of the underlying inference approach on SPARQL queries, supporting operators such as UNION in SPARQL. Such extensions may approach a limit when considering additional generic or type-level query constructs. For example, while we support a specific set of generic patterns in SPARQL, arbitrary constraints on type-level variables (i.e. variables that bind concepts) cannot be easily encoded in DL.

The query mapping itself also poses interesting questions: While in our work, we do not intend to execute the SPARQL `CONSTRUCT` queries, the mappings could inform implementations of G-CORE over SPARQL triple stores. To this end, the RDF reification would have to be investigated empirically regarding runtime and space performance and possibly adapted, given that our mapping favors a formal ease of use over efficiency concerns.

References

1 Ghadeer Abuoda, Daniele Dell'Aglio, Arthur Keen, and Katja Hose. Transforming rdf-star to property graphs: A preliminary analysis of transformation approaches. In *Proceedings of the QuWeDa 2022: 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs co-located with 21st International Semantic Web Conference (ISWC 2022), Hangzhou, China, 23-27 October 2022*, volume 3279 of *CEUR Workshop Proceedings*, pages 17–32. CEUR-WS.org, 2022. URL: `https://ceur-ws.org/Vol-3279/paper2.pdf`.

2 Lakshya A. Agrawal, Nikunj Singhal, and Raghava Mutharaju. A SPARQL to cypher transpiler: Proposal and initial results. In *CODS-COMAD 2022:*

5th Joint International Conference on Data Science & Management of Data (9th ACM IKDD CODS and 27th COMAD), Bangalore, India, January 8 - 10, 2022, pages 312–313. ACM, 2022. doi:10.1145/3493700.3493757.

3 Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A core for future graph query languages. In *Proc. of SIGMOD*, pages 1421–1432. ACM, 2018. doi:10.1145/3183713.3190654.

4 Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. Pg-schema: Schemas for property graphs. *Proc. ACM Manag. Data*, 1(2):198:1–198:25, 2023. doi:10.1145/3589778.

5 Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. RDF and property graphs interoperability: Status and issues. In *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción, Paraguay, June 3-7, 2019*, volume 2369 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019. URL: https://ceur-ws.org/Vol-2369/paper01.pdf.

6 Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. Mapping RDF databases to property graph databases. *IEEE Access*, 8:86091–86110, 2020. doi:10.1109/ACCESS.2020.2993117.

7 Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

8 Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. SHACL: A description logic in disguise. In *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings*, volume 13416 of *Lecture Notes in Computer Science*, pages 75–88. Springer, 2022. doi:10.1007/978-3-031-15707-3_7.

9 Iovka Boneva, Jérémie Dusart, Daniel Fernández-Álvarez, and José Emilio Labra Gayo. Shape designer for shex and SHACL constraints. In *Proc. of the ISWC 2019 Satellite Tracks co-located with ISWC 2019*, volume 2456 of *CEUR Workshop Proceedings*, pages 269–272. CEUR-WS.org, 2019. URL: https://ceur-ws.org/Vol-2456/paper70.pdf.

10 Iovka Boneva, Benoît Groz, Jan Hidders, Filip Murlak, and Slawek Staworko. Static analysis of graph database transformations. In Floris Geerts, Hung Q. Ngo, and Stavros Sintos, editors, *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 251–261. ACM, 2023. doi:10.1145/3584372.3588654.

11 Angela Bonifati, Stefania Dumbrava, and Nicolas Mir. Hierarchical clustering for property graph schema discovery. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, pages 2:449–2:453. OpenProceedings.org, 2022. doi:10.48786/EDBT.2022.39.

12 Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data Knowl. Eng.*, 68(10):973–1000, 2009. doi:10.1016/J.DATAK.2009.04.001.

13 Richard Cyganiak, David Wood, Markus Lanthaler, Graham Klyne, Jeremy J. Carroll, and Brian McBride. RDF concepts and abstract syntax, 2014. URL: https://www.w3.org/TR/rdf11-concepts/.

14 Thomas Delva, Birte De Smedt, Sitt Min Oo, Dylan Van Assche, Sven Lieber, and Anastasia Dimou. RML2SHACL: RDF generation taking shape. In *Proc. of Knowledge Capture Conference*, pages 153–160. ACM, 2021. doi:10.1145/3460210.3493562.

15 Wenfei Fan, Shuai Ma, Yanli Hu, Jie Liu, and Yinghui Wu. Propagating functional dependencies with conditions. *Proc. VLDB Endow.*, 1(1):391–407, 2008. doi:10.14778/1453856.1453901.

16 Daniel Fernández-Álvarez, José Emilio Labra Gayo, and Daniel Gayo-Avello. Automatic extraction of shapes using shexer. *Knowledge-Based Systems*, 238:107975, 2022. doi:10.1016/J.KNOSYS.2021.107975.

17 Benoît Groz, Aurélien Lemay, Slawek Staworko, and Piotr Wieczorek. Inference of shape graphs for graph databases. In *ICDT*, volume 220 of *LIPIcs*, pages 14:1–14:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICDT.2022.14.

18 Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 query language, 2013. URL: https://www.w3.org/TR/sparql11-query/.

19 Olaf Hartig. Foundations of RDF* and SPARQL* (an alternative approach to statement-level metadata in RDF). In *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017*, volume 1912 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017. URL: https://ceur-ws.org/Vol-1912/paper12.pdf.

20 Olaf Hartig. Foundations to query labeled property graphs using SPARQL. In *Joint Proceedings of the 1st International Workshop On Semantics For Transport and the 1st International Workshop on Approaches for Making Data Interoperable co-located with 15th Semantics Conference (SEMANTiCS 2019), Karlsruhe, Germany, September 9, 2019*, volume 2447 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019. URL: https://ceur-ws.org/Vol-2447/paper3.pdf.

21 ISO. Information technology – Database languages – GQL. Standard, International Organization for Standardization, Geneva, CH, 2024.

22 Barry E. Jacobs, Alan R. Aronson, and Anthony C. Klug. On interpretations of relational languages and solutions to the implied constraint problem.

ACM Trans. Database Syst., 7(2):291–315, 1982. doi:10.1145/319702.319730.

23 JTC1. GQL Database Language, 2024. URL: https://jtc1info.org/slug/gql-database-language.

24 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 '22*, pages 10:1–10:9. ACM, 2022. doi:10.1145/3534540.3534695.

25 Anthony C. Klug and Rod Price. Determining view dependencies using tableaux. *ACM Trans. Database Syst.*, 7(3):361–380, 1982. doi:10.1145/319732.319738.

26 Holger Knublauch and Dimitris Kontokostas. Shapes Constraint Language (SHACL), 2017. URL: https://www.w3.org/TR/shacl/.

27 Egor V. Kostylev, Juan L. Reutter, and Martín Ugarte. CONSTRUCT queries in SPARQL. In *Proc. of International Conference on Database Theory, ICDT*, volume 31 of *LIPIcs*, pages 212–229. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.ICDT.2015.212.

28 KRR Group at University of Oxford. Hermit OWL reasoner, 2008. URL: http://www.hermit-reasoner.com/.

29 Hanâ Lbath, Angela Bonifati, and Russ Harmer. Schema inference for property graphs. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, pages 499–504. OpenProceedings.org, 2021. doi:10.5441/002/EDBT.2021.58.

30 Martin Leinberger, Philipp Seifer, Claudia Schon, Ralf Lämmel, and Steffen Staab. Type checking program code using SHACL. In *Proc. of ISWC*, volume 11778 of *LNCS*, pages 399–417. Springer, 2019. doi:10.1007/978-3-030-30793-6_23.

31 Mohamed Nadjib Mami, Damien Graux, Harsh Thakkar, Simon Scerri, Sören Auer, and Jens Lehmann. The query translation landscape: a survey. *CoRR*, abs/1910.03118, 2019. arXiv:1910.03118.

32 Nandana Mihindukulasooriya, Mohammad Rifat Ahmmad Rashid, Giuseppe Rizzo, Raúl García-Castro, Óscar Corcho, and Marco Torchiano. RDF shape induction using knowledge base profiling. In *Prov. of the Symposium on Applied Computing*, pages 1952–1959. ACM, 2018. doi:10.1145/3167132.3167341.

33 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) co-located with ISWC 2019*, volume 2599 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019. URL: https://ceur-ws.org/Vol-2599/CKG2019_paper_4.pdf.

34 Pouya Ghiasnezhad Omran, Kerry Taylor, Sergio José Rodríguez Méndez, and Armin Haller. Learning SHACL shapes from knowledge graphs. *Semantic Web*, 14(1):101–121, 2023. doi:10.3233/SW-223063.

35 Rossana Paciello, Luca Trani, Daniele Bailo, Valerio Vinciarelli, and Manuela Sbarra. Shapeness: A shacl-driven metadata editor. In *Metadata and Semantic Research - 16th Research Conference, MTSR 2022, London, UK, November 7-11, 2022, Revised Selected Papers*, volume 1789 of *Communications in Computer and Information Science*, pages 274–288. Springer, 2022. doi:10.1007/978-3-031-39141-5_23.

36 Eric Prud'hommeaux, José Emilio Labra Gayo, and Harold R. Solbrig. Shape expressions: an RDF validation and transformation language. In *Proc. of the International Conference on Semantic Systems, SEMANTiCS*, pages 32–40. ACM, 2014. doi:10.1145/2660517.2660523.

37 Kashif Rabbani, Matteo Lissandrini, Angela Bonifati, and Katja Hose. Transforming RDF graphs to property graphs using standardized schemas. *Proc. ACM Manag. Data*, 2(6):242:1–242:25, 2024. doi:10.1145/3698817.

38 Kashif Rabbani, Matteo Lissandrini, and Katja Hose. Extraction of validating shapes from very large knowledge graphs. *Proc. VLDB Endow.*, 16(5):1023–1032, 2023. doi:10.14778/3579075.3579078.

39 Mariano Rodriguez-Muro and Martín Rezk. Efficient SPARQL-to-SQL with R2RML mappings. *J. Web Semant.*, 33:141–169, 2015. doi:10.1016/J.WEBSEM.2015.03.001.

40 Philipp Seifer. softlang/s2s. Software, swhId: swh:1:dir:a38354c2cab3efb8abdc9e80f611d19aa3531d82 (visited on 2026-08-17). URL: https://github.com/softlang/s2s, doi:10.4230/artifacts.27685.

41 Philipp Seifer, Daniel Hernández, Ralf Lämmel, and Steffen Staab. From shapes to shapes: Inferring SHACL shapes for results of SPARQL CONSTRUCT queries. In *Proceedings of the ACM Web Conference 2024, (WWW '24)*. ACM, 2024. doi:10.1145/3589334.3645550.

42 Philipp Seifer, Daniel Hernández, Ralf Lämmel, and Steffen Staab. Code for From Shapes to Shapes (V1), 2024. URL: https://darus.uni-stuttgart.de/dataset.xhtml?persistentId=doi:10.18419/DARUS-3977&version=1.0.

43 Philipp Seifer, Daniel Hernández, Ralf Lämmel, and Steffen Staab. Code for From Shapes to Shapes (V2), 2024. doi:10.18419/darus-3977.

44 Philipp Seifer, Ralf Lämmel, and Steffen Staab. Progs: Property graph shapes language. In *ISWC 2021*, volume 12922 of *LNCS*, pages 392–409. Springer, 2021. doi:10.1007/978-3-030-88361-4_23.

45 Blerina Spahiu, Andrea Maurino, and Matteo Palmonari. Towards improving the quality of knowledge graphs with data-driven ontology patterns and SHACL. In *Proc. of the Workshop on Ontology Design and Patterns (WOP 2018) co-located with ISWC 2018*, volume 2195 of *CEUR Workshop Proceedings*, pages 52–66. CEUR-WS.org, 2018. URL: https://ceur-ws.org/Vol-2195/research_paper_2.pdf.

46 Benjamin A. Steer, Alhamza Alnaimi, Marco A. B. F. G. Lotz, Félix Cuadrado, Luis M. Vaquero, and Joan Varvenne. Cytosm: Declarative property graph queries without data migration. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL,*

USA, May 14 - 19, 2017, pages 4:1–4:6. ACM, 2017. doi:10.1145/3078447.3078451.

47 Michael Stonebraker. Implementation of integrity constraints and views by query modification. In *Proc. of SIGMOD*, pages 65–78. ACM, 1975. doi:10.1145/500080.500091.

48 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), Houston, TX, USA, June 10, 2018*, pages 12:1–12:5. ACM, 2018. doi:10.1145/3210259.3210271.

49 Ratan Bahadur Thapa and Martin Giese. A source-to-target constraint rewriting for direct mapping. In *Proc. of ISWC*, volume 12922 of *LNCS*, pages 21–38. Springer, 2021. doi:10.1007/978-3-030-88361-4_2.

50 Ratan Bahadur Thapa and Martin Giese. Optimizing SPARQL queries with SHACL. In *The Semantic Web - ISWC 2023 - 22nd International Semantic Web Conference, Athens, Greece, November 6-10, 2023, Proceedings, Part I*, volume 14265 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2023. doi:10.1007/978-3-031-47240-4_3.

51 Dominik Tomaszuk, Renzo Angles, and Harsh Thakkar. PGO: describing property graphs in RDF. *IEEE Access*, 8:118355–118369, 2020. doi:10.1109/ACCESS.2020.3002018.

52 Zihao Zhao, Xiaodong Ge, Zhihong Shen, Chuan Hu, and Huajin Wang. S2ctrans: Building a bridge from SPARQL to cypher. In *Database and Expert Systems Applications - 34th International Conference, DEXA 2023, Penang, Malaysia, August 28-30, 2023, Proceedings, Part I*, volume 14146 of *Lecture Notes in Computer Science*, pages 424–430. Springer, 2023. doi:10.1007/978-3-031-39847-6_33.