PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

THE EXPRESSIVENESS OF SHACL AND A TRACTABLE LANGUAGE FRAGMENT PROPOSAL

FERNANDO ALBERTO FLORENZANO HERNÁNDEZ

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
JUAN REUTTER

Santiago de Chile, May 2020

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

THE EXPRESSIVENESS OF SHACL AND A TRACTABLE LANGUAGE FRAGMENT PROPOSAL

FERNANDO ALBERTO FLORENZANO HERNÁNDEZ

Members of the Committee:
JUAN REUTTER
CRISTIAN RIVEROS
JORGE PÉREZ
CRISTIAN ESCAURIAZA

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, May 2020

Gratefully to my family, friends,
teachers and myself.

ACKNOWLEDGEMENTS

I would like to thank a lot of people, so bear with me. Thanks to:

My dear family. My parents Victoria and Fernando, for dealing with me, loving me, and always supporting me in every way. Also my loving siblings, for being present and helping me grow. Specially my dear sisters Victoria and Andrea, for being exceptionally strong and good women. Everyday I try to live following your example.

My dear friends and loved ones that I met along the way and have stuck by. Meeting every single one of your has given me something that I cherish and will always remember. We may not see each other as much, but you truly are my chosen family, the ones I'll always want around for the special moments. Let's have coffee some time please.

The 2019 Advanced Programming course team. Being a teacher was an unexpectedly emotional and intense experience that made me grow during my investigation, and was great thanks to the awesome people that I worked with. Thanks to Daniela, Dante, Enzo, Hernán, Ignacio, Cristian, Antonio, Vicente and all the other 54 teaching assistants that were part of the teaching team during that year. Also, thanks to my dear students, all 162, who taught me in a lot of ways too and helped me improve myself even more.

All the people that I met at the Computer Science Department at PUC: staff, teachers, and fellow students. You taught me everything I know now about CS, and helped to create a second home for me during all these years.

Cristian Riveros, one of my teachers. You taught me, you gave me opportunities to work as a TA, to go into investigation, and to become a teacher. But most importantly, you saw something in me, helped me to make the most of it and constantly inspired me as an example and role model.

Juan Reutter, one of my teachers and my supervisor. Even though our working styles were very different, times got tough and sometimes I wanted to give up, we made it through. You did an incredible job as a supervisor, but more importantly, you helped me in amazing

ways to grow as a professional, as a person, and to see the world in a different way. I consider our experience together as important as this thesis.

Myself, for hanging on and asking for help when needed. It took a while, but I am improving my mental health and I know me better than ever. You go girl.

And finally, Ariana Grande. Thanks, just for existing. You deserved the Grammy.

Thanks again, to every single one of you, you made me who I am today.

Love, Fernando.

Ugh, I'm crying.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

SHACL (Shapes Constraint Language) is a specification for describing and validating RDF graphs that has recently become a W3C recommendation. The main difficulty with its use is the absence of guidelines about the way recursive constraints should be handled. In addition to that, is the fact that RDF graphs are often exposed as SPARQL endpoints, and therefore only accessible via queries, which makes validation depend on this systems. In this thesis, we extend previous work with the objective of providing a better understanding of the validation problem as a whole. We first investigate the possibility of validating a graph against non-recursive constraints through all in-memory processing and alternatively through a single query evaluation. Then for the recursive case, since the problem has been shown to be NP-hard, we review the known fragment hierarchy and their challenges. Finally, we propose a new fragment of SHACL schemas that contain previously known groups and show an algorithm to evaluate efficiently this new fragment. The latter, can be used when dealing with recursive but tractable fragments of SHACL, without the need for an external help.

Keywords: SHACL, RDF validation, RDF schemas, recursive SHACL, logic

RESUMEN

SHACL (Shapes Constraint Language) es una especificación para describir y validar grafos RDF que recientemente se convirtió en recomendación de la W3C. La dificultad principal que presenta su uso es la ausencia de una definición oficial para el manejo de restricciones recursivas. Además, el hecho de que grafos RDF por lo general son accesibles mediante alojamiento remoto a través de solo consultas SPARQL hace que la validación dependa de dichos sistemas. En esta tesis, extendemos trabajo previo con el objetivo de mejorar el entendimiento de lo conocido del problema de validación. Primero, investigamos la posibilidad de validar un grafo contra esquemas no recursivos utilizando solo procesamiento en memoria, y mediante el procesamiento de una única consulta general. Para el caso recursivo, cuyo problema es NP-duro, revisamos la jerarquía de fragmentos de SHACL conocidos y sus respectivas dificultades. Finalmente, proponemos un nuevo fragmento de restricciones y mostramos un algoritmo que resuelve eficientemente el problema de validación. Este último se puede utilizar cuando es necesario manejar restricciones recursivas, pero manteniendo cotas de ejecución eficientes sin tener que recurrir a maquinaria externa.

Palabras Claves: SHACL, validación RDF, esquemas RDF, SHACL recursivo, lógica

1. INTRODUCTION

RDF (for Resource Description Framework),[1] is a model for data interchange on the Web that organizes data as a directed and labeled *graph*. It is based on the idea of making statements about resources (in particular web resources) in expressions of the form subject–predicate–object, known as triples. The subject denotes the resource, and the predicate denotes traits or aspects of the resource, and expresses a relationship between the subject and the object. Thus, the statements that form a RDF database can be seen as nodes and edges declarations in a graph, where the subject and object are nodes, while the predicate is a labeled edge between subject and object. This data model enables the encoding of semantics into web databases, and therefore helps to make the internet data computer-readable. The latter is the main goal of the Semantic Web, the proposal to make the web that can be processed by machines, and uses RDF as one of its key technologies.

The success of RDF was largely due the fact that it can be easily published and queried without bounding to a specific schema (Berners-Lee, Hendler, & Lassila, 2001). But RDF over time has turned into more than a simple data exchange format (Arenas, Gutiérrez, & Pérez, 2009), and a key challenge for current RDF-based applications is checking quality (correctness and completeness) of a dataset. Several systems already provide facilities for RDF validation (see e.g. (Ekaputra & Lin, 2016)), including commercial products (*Stardog ICV*, n.d.; *TopBraid Composer*, n.d.). This created a need for standardizing a declarative language for RDF constraints, and for formal mechanisms to detect and describe violations of such constraints.

SHACL (for SHApes Constraint Language),[2] is an expressive constraint language for RDF graph, which became a W3C recommendation in 2017. The idea of SHACL is to group constraints in so-called "shapes" to be verified by "target nodes" in the graph under validation, where shapes can even reference each other. Consequently, SHACL *schema* can also be seen as a way of describing a set of RDF graphs; precisely those that can be validated against it.

[1] `https://www.w3.org/rdf/`
[2] `https://www.w3.org/TR/shacl/`

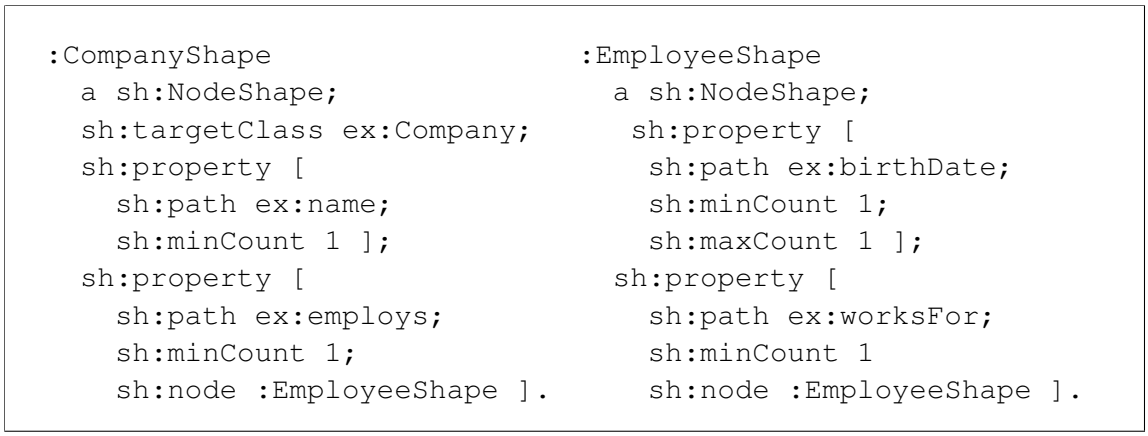

FIGURE 1.1. Two SHACL shapes, about companies and employees

Figure 1.1 presents two simple SHACL shapes. The left one, called `:CompanyShape`, is meant to verify constraints over company entities in a database. The second triple, `:CompanyShape sh:targetClass ex:Company`, is the *target definition* of the shape, and specifies that all instances of the class `ex:Company` must conform to this shape. These are called the *target nodes* of a shape. The next triples specify the constraints that must be satisfied by such nodes, namely that they must have at least one name, and at least one employee (i.e. their `ex:employs`-successors in the graph), that must conform to the shape `:EmployeeShape`.

The rightmost shape, called `:EmployeeShape`, is meant to define employees in this example. It does not have a target definition (therefore no target node either), and states that an employee must have exactly one birth date, and work for at least one entity that also conforms to the shape `:EmployeeShape`.

A key feature of SHACL is the possibility for a shape to refer to another or to itself (like `:EmployeeShape` refers to itself for instance). This allows designing schemas in a modular fashion, but also reusing existing shapes in a new schema, thus favoring semantic interoperability.

The SHACL specification provides semantics for *graph validation*, i.e. what it means for a graph to conform to a set of shapes: a graph is *valid* against a set of shapes if each target node of each shape satisfies the constraints associated to it. If these constraints

contain shape references, (e.g. companies requiring employees in the example above), then the propagated constraints (to neighbors, neighbors of neighbors, etc.) must be satisfied as well.

Unfortunately, the SHACL specification leaves explicitly undefined the semantics of validation for schemas with circular references (called *recursive* below), such as the one of Figure 1.1, where `:EmployeeShape` refers to itself. Such schemas can be expected to appear in practice though, either by design (e.g. to characterize relations between events, or a structure of unbounded size, such as a tree), or as a simple side-effect of the growth of the number of shapes.

Semantics for graph validation against possibly recursive shapes was later proposed in (Corman, Reutter, & Savkovic, 2018a) (for the so-called "core constraint components" of the SHACL specification) that complies with the specification in the non-recursive case. Based on to this semantics, the first graph in Figure 1.2 is valid against the shapes of Figure 1.1. The second graph is not, because `ex:Mark` does not work for another entity that satisfies `:EmployeeShape`. The third graph is trivially valid, since there is no target node to initiate validation.

Negation is another important feature of the SHACL specification (allowing for instance to state that a node cannot conform to two given shapes at the same time, or to express functionality, like "exactly one birth date for employees" in Figure 1.1). But as shown in (Corman et al., 2018a), the interplay between recursion and negation makes the graph validation problem significantly more complex (NP-hard in the size of the graph).

As SHACL is gaining traction, more validation engines become available.[3] However, guidance about the way graph validation may be implemented is still lacking. In particular, as far as we are aware all existing implementations deal with recursive schemas in their own terms, without a principled approach to handle the interplay between recursion and negation. Even tough some well behaved interplays were identified in (Corman et al.,

[3] `https://w3c.github.io/data-shapes/data-shapes-test-suite/`

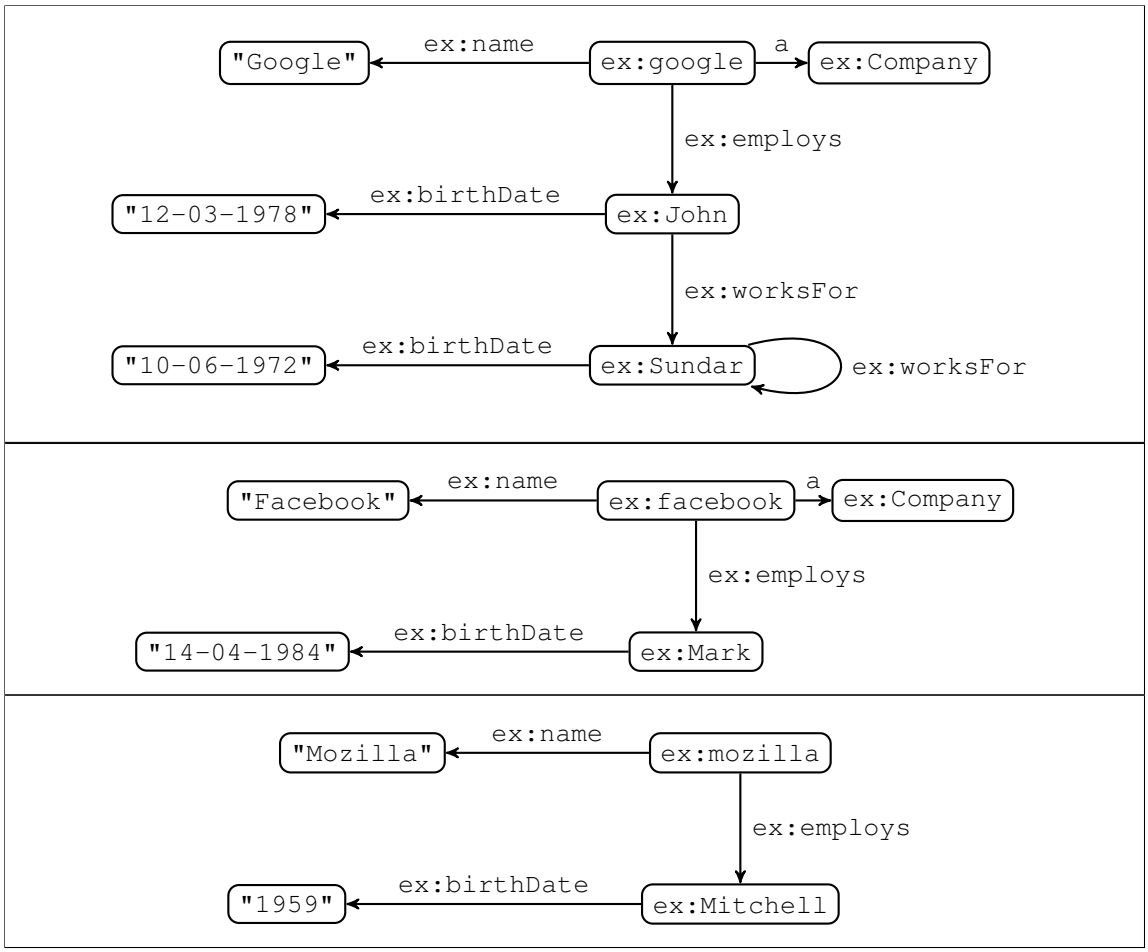

FIGURE 1.2. RDF graphs. Top and bottom ones are valid against the shapes of Figure 1.1, while the middle one is invalid.

2018a), it failed to define a general case of how could recursion and negation be used together.

Another key aspect of graph validation is the way the data can be accessed. RDF graphs are generally exposed as SPARQL endpoints, i.e. primarily (and sometimes exclusively) accessible via SPARQL queries. This is often the case for large graphs that may not fit into memory, exposed via triple stores. Therefore an important feature of a SHACL validation engine is the possibility to check conformance of a graph by issuing SPARQL queries over it. This may also be needed when integrating several data sources not meant to be materialized together, or simply to test conformance of data that one does not own.

Several engines can already perform validation via SPARQL queries for fragments of SHACL. But there is no clear shared guideline between engines as how to approach the problem and are solved independently.

Furthermore, none of these engines, to our knowledge, tackle the problem in the presence of recursive constraints.[4] This should not come as a surprise: as it was shown in (Corman, Florenzano, Reutter, & Savkovic, 2019) that recursive shapes go beyond the expressive power of SPARQL. This makes validation via SPARQL queries significantly more involved: if the schema is recursive, it is not possible in general to retrieve target nodes violating a given shape by issuing a single SPARQL query. This means that some extra computation (in addition to SPARQL query evaluation) needs to be performed, in memory.

This thesis provides a theoretical investigation of graph validation against SHACL schemas that may be recursive and present negation, as well as considering graphs that are only accessible via SPARQL queries or in-memory. First, we show different approaches to validation for non-recursive schemas: two in-memory algorithms as well as an online approach performed via SPARQL queries.

Then, we define a recursive fragment that along with allowing some in-memory computation, while still accessing the endpoint via queries only, produces a tractable algorithm for a more general case. For the proposed recursive fragment we show that an algorithm that performs propositional inference on the fly can efficiently solve the validation problem.

The first half of the results presented in this thesis are also presented in (Corman et al., 2019), while the second half is an extension and reviewed version of previous results with completely new aspects, as for the new proposed tractable fragment.

[4] with the exception of *Shaclex* (*Shaclex*, n.d.), which can handle recursion, but not recursion and negation together in a principled way.

Organization. Section 2 introduces the original SHACL validation problem and the necessary logic model used in this thesis. Section 3 studies different approaches for the non-recursive case, whereas Sections 4, 5 and 6 focus on the recursive case of the problem and proposes a recursive but tractable fragment. Section 7 discusses conclusions, future work and perspectives.

2. PRELIMINARIES

In this section we provide an overview of the constraint validation mechanism described in the SHACL specification, discuss both its non-recursive case and its recursive constraints case, and introduce the abstract syntax this thesis uses to model the SHACL validation problem. Before explaining the SHACL specification and notation, we introduce both the RDF specification and the query language SPARQL to new readers.

RDF. As mentioned earlier, RDF is a model and format to store data, based on the idea of making statements about resources in expressions of the form subject–predicate–object, known as triples. These triples can be modeled as edge definitions in a graph, so if we abstract away from the concrete RDF syntax, an RDF graph $\mathcal{G}$ can be defined as: a labeled oriented graph $\mathcal{G} = \langle V_{\mathcal{G}}, E_{\mathcal{G}} \rangle$, where $V_{\mathcal{G}}$ is the set of nodes of $\mathcal{G}$, and $E_{\mathcal{G}}$ is a set of triples (edges) of the from (v_1, p, v_2), meaning that there is an edge in $\mathcal{G}$ from v_1 to v_2 labeled with property p.

As an example, Figure 2.1 shows what would be the RDF triples that correspond to the three graphs presented in Figure 1.2. Some nodes may be resources with multiple outgoing edges that describes them, while other might be object literals as integers and strings used to represent the latter nodes. In Figure 2.1, the triple `ex:John ex:birthDate "12-03-1978"` describes the resource (and node) `ex:John`, specifying it's birth date (edge) with the literal (and also node) `"12-03-1978"`. While, the triple `ex:John ex:worksFor ex:Sundar` describes a relationship between the two resources (and nodes) `ex:John` and `ex:Sundar`, by stating that the subject works for the object (edge relationship).

SPARQL. On the other hand, SPARQL is a semantic query language for databases able to retrieve and manipulate data stored in RDF format. SPARQL queries mainly consists of RDF triple patterns, but also conjunctions, disjunctions, and even filtering (also through triple patterns). Figure 2.2 shows two SPARQL queries that could be applied on our previous example.

```
ex:google a ex:Company .
ex:google ex:name "Google" .
ex:google ex:employs ex:John .
ex:John ex:birthDate "12-03-1978" .
ex:John ex:worksFor ex:Sundar .
ex:Sundar ex:birthDate "10-06-1972" .
ex:Sundar ex:worksFor ex:Sundar .
```

```
ex:facebook a ex:Company .
ex:facebook ex:name "Facebook" .
ex:facebook ex:employs ex:Mark .
ex:Mark ex:birthDate "12-03-1978" .
```

```
ex:mozilla ex:name "Mozilla" .
ex:mozilla ex:employs ex:Mitchell .
ex:Mitchell ex:birthDate "1959" .
```

FIGURE 2.1. Three graphs from Figure 1.2 as RDF triples.

The top query exemplifies a simple SPARQL query. As a result it returns answers consisting of two variables: `?x` and `?y`; by assigning resources in the target RDF dataset to each variable if they satisfy certain conditions. These conditions are expressed in the `WHERE` clause, where two triple patterns are needed to be matched in order to satisfy the query. In this case, the query would match resources that are described as `ex:Company`, and also have a name; and would return both the matching resource in `?x` and the matched name in `?y`. In the example, the query on the first RDF store returns `?x` $\mapsto$ `ex:google`, `?y` $\mapsto$ `"Google"`, while on the second `?x` $\mapsto$ `ex:facebook`, `?y` $\mapsto$ `"Facebook"`. In the third one an empty answer is returned since `ex:mozilla` does not match the triple pattern specified in the query, since the `ex:Company` relationship is missing.

The bottom query also applies a triple pattern, but adds a `FILTER NOT EXISTS` clause that makes the main pattern match if it does not match with the `?z ex:employs ?x` pattern. Therefore, this query returns all resources with a birth date, but does not work for any other resource. In the example, the query on the second RDF store returns `?x` $\mapsto$ `ex:Mark`, on the third `?x` $\mapsto$ `ex:Mitchell`, while on the first one an empty answer is returned since the are no resources that match the query.

```
SELECT ?x ?y
WHERE {
    ?x a ex:Company.
    ?x ex:name ?y.
}

SELECT ?x
WHERE {
    ?x ex:birthDate ?y.
    FILTER NOT EXISTS {?x ex:worksFor ?z}
}
```

FIGURE 2.2. Two SPARQL queries that could be applied over the RDF stores in Figure 2.1.

We use $[\![Q]\!]^{\mathcal{G}}$ to denote the evaluation of a SPARQL query Q over an RDF graph $\mathcal{G}$. This evaluation is given as a set of *solution mappings*, each of which maps variables of Q to nodes of $\mathcal{G}$. All solution mappings considered in this thesis are *total* functions over the variables projected by Q. We use $\{?x_1 \mapsto v_1, \ldots, ?x_n \mapsto v_n\}$ to denote the solution mapping that maps $?x_i$ to v_i for $i \in [1, \ldots, n]$. However, if Q is a *unary* query (i.e. if it projects only one variable), we may also represent $[\![Q]\!]^{\mathcal{G}} = \{\{?x \mapsto v_1\}, \ldots, \{?x \mapsto v_m\}\}$ as the set of nodes $\{v_1, \ldots, v_m\}$.

SHACL. As already mentioned, SHACL is a constraint language for RDF stores. Constraints are established by defining a set of shapes also in a triple-like pattern fashion, each with certain properties that a set of target nodes must conform to. Figure 1.1 showed a taste of how these shapes can be defined and express, but many other features exists to establish shapes.

An important one is the possibility to declare negated constraints. For instance in Figure 2.3, shape `:TraineeShape` describes a trainee as someone with exactly one birth date that works for a boss, while the shape `:BossShape` uses `sh:not` to describe a boss as someone with exactly one birth date but that is not a trainee. In this case, the constraint for `:TraineeShape` will hold for the target node `ex:Fernando` if a node successor via property `ex:worksFor` violates the constraints for `:TraineeShape`.

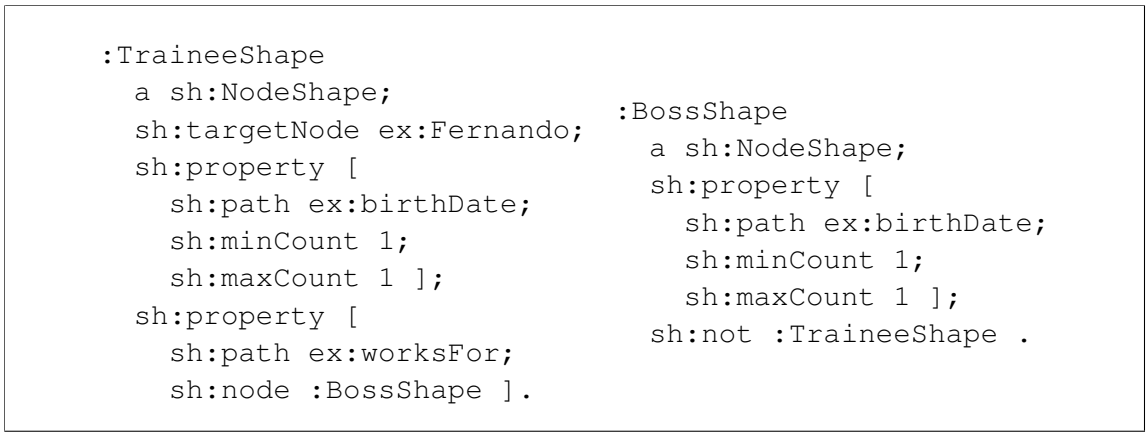

FIGURE 2.3. Shape definition for trainees and bosses that uses negation.

This thesis follows the abstract syntax for SHACL core constraint components introduced in (Corman et al., 2018a), and also later used in (Corman et al., 2019). In the following, we review this syntax and the associated semantics for graph validation.

A *shape schema* $\mathcal{S}$ is represented as a triple $\langle S, \mathrm{targ}, \mathrm{def}\rangle$, where S is a set of *shape names*, targ is a function that assigns a *target query* to each $s \in S$, and def is a function that assigns a *constraint* to each $s \in S$.

For each $s \in S$, $\mathrm{targ}(s)$ is a unary query, which can be evaluated over the graph under validation in order to retrieve the *target nodes* of s. The SHACL specification only allows target queries with a limited expressivity, but for the purpose of this thesis, $\mathrm{targ}(s)$ can be assumed to be an arbitrary unary SPARQL query. If a shape has no target definition (like the shape `:BossShape` in Figure 2.3), we use an arbitrary empty SPARQL query (i.e. with no answer, in any graph), denoted with $\perp$, meaning there is no target nodes for that shape. Note that does not mean the shape is not used, as it may be referenced by other shapes.

The constraint $\mathrm{def}(s)$ for shape s is represented as a formula ϕ verifying the following grammar:

$$\phi \ ::= \ \top \mid s \mid I \mid \phi \wedge \phi \mid \neg\phi \mid \ \geq_n r.\phi \mid \mathrm{EQ}(r_1, r_2)$$

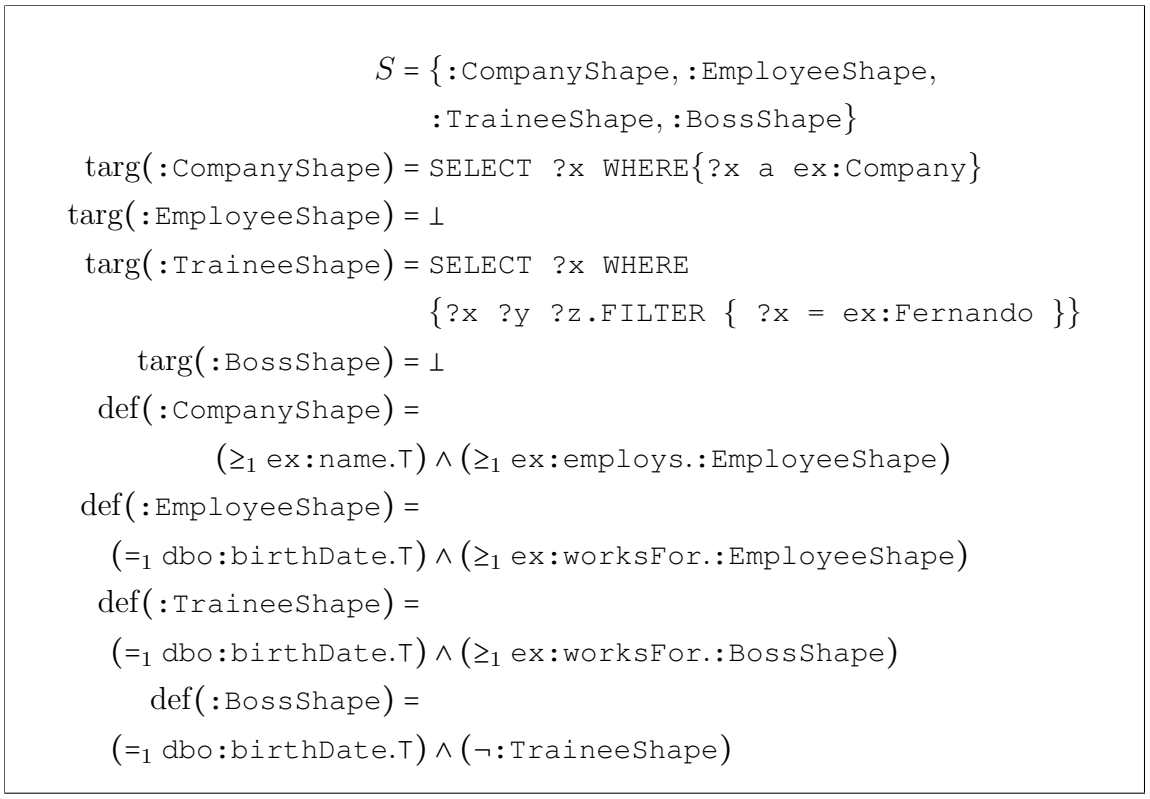

FIGURE 2.4. Abstract representation for shape schema defined in Figure 1.1 and extended with shapes from Figure 2.3.

where s is a shape name, I is an IRI,[1] r is a SHACL path[2], and $n \in \mathbb{N}^+$. Syntactic sugar can be considered to denote $\phi_1 \vee \phi_2$ for $\neg(\neg\phi_1 \wedge \neg\phi_2)$, $\leq_n r.\phi$ for $\neg(\geq_{n+1} r.\phi)$, and $=_n r.\phi$ for $(\geq_n r.\phi) \wedge (\leq_n r.\phi)$. A translation from SHACL core constraint components to this grammar and conversely can be found in (Corman, Reutter, & Savkovic, 2018b). As an example, Figure 2.4 shows the translation of the previous example into the introduced abstract representation.

Checking whether a graph is valid against a set of shapes may be viewed as a two-step process. The first step consists in iterating over all shapes, and retrieve their respective target nodes in the graph. The limited expressivity the target language has allows all targets

[1]More exactly, I is an abstraction, standing for any syntactic constraint over an RDF term: exact value, data type, regex, etc.
[2]SHACL paths are built like SPARQL property paths, but without the *NegatedPropertySet* operator

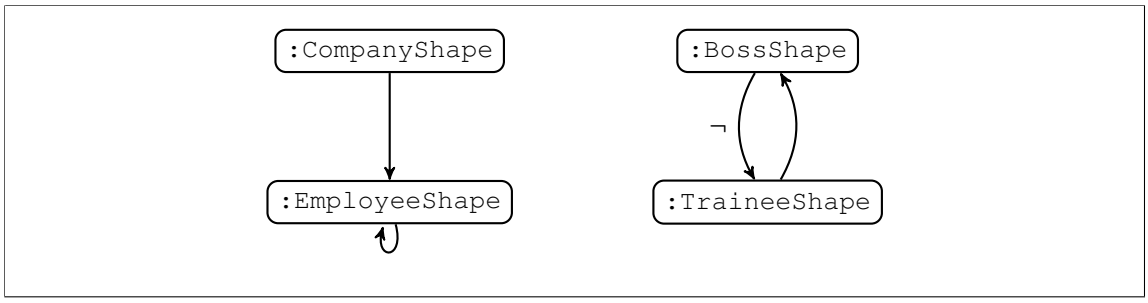

FIGURE 2.5. Dependency graph for schema in Figure 2.4.

of a shape in the graph to be retrieved before constraint validation in polynomial time in data complexity, meaning in terms of the size of the graph.

The second step consists in iterating over each target node of each shape, and check whether the node satisfies the corresponding constraint. Some of the constraints may be validated by looking locally at the graph, i.e. at the IRI of a node and its outgoing paths. But some others may also trigger a recursive call, over the constraint of another shape referenced by the initial one, to be checked for even other nodes that are not target nodes. This checking may also do another recursive call, and so on.

The recursive constraint calling is not much of a problem when if it stops eventually and constraints for target node can be validated and answered. This is the case for what we call later the non-recursive case: when the referencing between shapes is not cyclical. A problem arises when considering referencing cycles between shapes, which would make the previous description run without end since recursive calls could not end.

It also gets even trickier and unclear when considering negation in referencing cycles. Situations can arise where constraint satisfaction as a boolean decision is not enough. For instance, consider the graphs in 2.6 to be validated with our example schema from Figure 2.3. The top graph is valid against our set of shapes, but the bottom graph is not.

If `ex:Fernando` is considered to satisfy `:TraineeShape`, then is because itself also satisfies `:BossShape`, but this contradicts its own definition. Note that this is even stronger, since if `ex:Fernando` is assigned to not be of `:TraineeShape`, this would satisfy `:BossShape`, but then the conditions for `:TraineeShape` are met. Therefore,

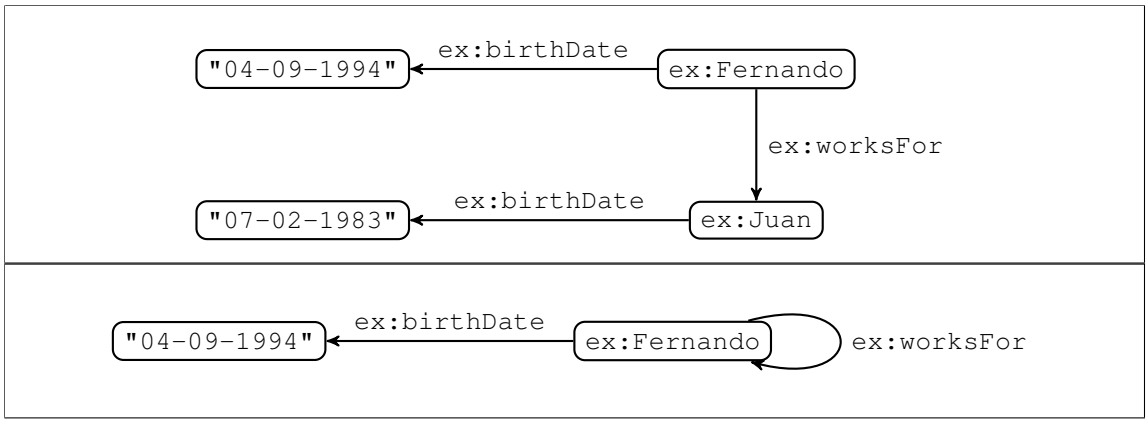

FIGURE 2.6. Graphs to be validated considering the schema in Figure2.3.

the idea of assigning shapes in only two options is not sufficient for these cases, and a third option that assigns neither the shape or its negation is needed. This is understood as *partial* assignments, which changes the validation problem and arises its own flavor for it. The regular option of always assigning either a shape or its negation is known as *total*.

Semantics. Since the semantics for recursive schemas is left undefined in the SHACL specification, we use the framework proposed in (Corman et al., 2018a). The evaluation of a formula is defined with respect to a given assignment, i.e. intuitively a labeling of the nodes of the graph with sets of shape names.

Formally, an *assignment* σ for a graph $\mathcal{G}$ and a schema $\mathcal{S} = \langle S, \text{targ}, \text{def} \rangle$ can be represented as a set of atoms of the form $s(v)$ or $\neg s(v)$, with $s \in S$ and $v \in V_{\mathcal{G}}$, that does not contain both $s(v)$ and $\neg s(v)$ for any $s \in S$ or $v \in V_{\mathcal{G}}$. An assignment σ is *total* if for every $s \in S$ and $v \in V_{\mathcal{G}}$, one of $s(v)$ or $\neg s(v)$ belongs to σ. Otherwise (if there are s and v such that neither $s(v)$ not $\neg s(v)$ belong to σ), the assignment is *partial*.

The semantics of a constraint ϕ is given in terms of a function $[\phi]^{\mathcal{G},v,\sigma}$, for a graph $\mathcal{G}$, node v and assignment σ. This function evaluates whether v satisfies ϕ given σ. This semantics depends on which type of assignments is considered. If we consider *total* assignments, then $[\phi]^{\mathcal{G},v,\sigma}$ is always true (1) or false (0), but when considering *partial* assignments a third option unknown is used (½). To be complete, we provide in Table 2.1 the

$$
\begin{aligned}
[\top]^{\mathcal{G},v,\sigma} &= 1 \\
[\neg\phi]^{\mathcal{G},v,\sigma} &= 1 - [\phi]^{\mathcal{G},v,\sigma} \\
[\phi_1 \wedge \phi_2]^{\mathcal{G},v,\sigma} &= \min\{[\phi_1]^{\mathcal{G},v,\sigma}, [\phi_2]^{\mathcal{G},v,\sigma}\} \\
[\mathrm{EQ}(r_1, r_2)]^{\mathcal{G},v,\sigma} &= \begin{cases} 1, \text{ if } \{v' \mid (v,v') \in [\![r_1]\!]^{\mathcal{G}}\} = \{v' \mid (v,v') \in [\![r_2]\!]^{\mathcal{G}}\} \\ 0 \text{ otherwise} \end{cases} \\
[I]^{\mathcal{G},v,\sigma} &= \begin{cases} 1, \text{ if } v \text{ is the IRI } I \\ 0 \text{ otherwise} \end{cases} \\
[s]^{\mathcal{G},v,\sigma} &= \begin{cases} 1, \text{ if } s(v) \in \sigma \\ 0, \text{ if } \neg s(v) \in \sigma \\ 1/2 \text{ otherwise} \end{cases} \\
[\geq_n r.\phi]^{\mathcal{G},v,\sigma} &= \begin{cases} 1, \text{ if } |\{v' \mid (v,v') \in [\![r]\!]^{\mathcal{G}} \text{ and } [\phi]^{\mathcal{G},v',\sigma} = 1\}| \geq n \\ 0, \text{ if } |\{v' \mid (v,v') \in [\![r]\!]^{\mathcal{G}}\}| - \\ \qquad\qquad |\{v' \mid (v,v') \in [\![r]\!]^{\mathcal{G}} \text{ and } [\phi]^{\mathcal{G},v',\sigma} = 0\}| < n \\ 1/2 \text{ otherwise} \end{cases}
\end{aligned}
$$

TABLE 2.1. Evaluation of constraint ϕ at node v in graph $\mathcal{G}$ given total assignment σ. We use $(v, v') \in [\![r]\!]^{\mathcal{G}}$ to say that v and v' are connected via SHACL path r.

partial semantics. This definition can be easily simplified to consider total assignments, and can be found (Corman et al., 2019).

Validation problem. A graph $\mathcal{G}$ satisfies a schema $\mathcal{S}$ if there is a way to assign shapes names to nodes of $\mathcal{G}$ such that all targets and constraints in $\mathcal{S}$ are satisfied. Since we consider two kinds of assignments (total and partial), we also define two types of validation.

Specifically, a graph $\mathcal{G}$ is *valid* against a shape schema $\mathcal{S} = \langle S, \mathrm{targ}, \mathrm{def}\rangle$ with respect to total (resp. partial) assignments if and only if there is a total (resp. partial) assignment σ for $\mathcal{G}$ and $\mathcal{S}$ that verifies the following, for each shape name $s \in S$:

- $s(v) \in \sigma$ for each node v in $[\![\mathrm{targ}(s)]\!]^{\mathcal{G}}$, and
- if $s(v) \in \sigma$, then $[\mathrm{def}(s)]^{\mathcal{G},v,\sigma} = 1$, and if $\neg s(v) \in \sigma$, then $[\mathrm{def}(s)]^{\mathcal{G},v,\sigma} = 0$.

The first condition ensures that all targets of a shape are assigned this shape, and the second condition that the assignment is consistent with respect to shape constraints. Any

assignment σ that verifies the previous conditions, is called *faithful* against the schema and graph.

We note that a total assignment is a specific case of partial assignment. So if $\mathcal{G}$ is valid against $\mathcal{S}$ with respect to total assignments, it is also valid with respect to partial assignments. The converse does not necessarily hold though. But as seen in (Corman et al., 2018b), there are several fragments for which this is true, and that holds for all the tractable fragments considered in this thesis. We use this property several times in the following sections.

Also as previous work, we describe schemas based on how shapes relate between each other. The *dependency graph* $G_{\mathcal{S}}$ of a schema $\mathcal{S} = \langle S, \mathrm{targ}, \mathrm{def} \rangle$ is a graph whose nodes are S, and such that there is an edge from s_1 to s_2 if and only if s_2 appears in $\mathrm{def}(s_1)$. This edge is called *negative* if such reference is in the scope of at least one negation, and *positive* otherwise. A schema is *recursive* if its dependency graph contains a cycle, and *stratified* if the dependency graph does not contain a cycle with at least one negative edge. In Figure 2.5, we see that the example schema is recursive, since `:EmployeeShape` references itself, and is not stratified since a negative cycle is present between `:TraineeShape` and `:BossShape`.

3. VALIDATION FOR NON-RECURSIVE SHACL

In this section we tackle first the simplest fragment for SHACL schemas, non-recursive schemas, and analyze two types of approaches to handle their corresponding validation problem: first in-memory approaches and then an online approach.

By in-memory approaches we mean ways to give an answer to the problem through algorithmic solutions supposing we have access to the database in the main memory of a machine, and therefore leaving all computations to the current working machine. Whereas by online approach we mean to give a solution by querying an endpoint that may be foreign to the working machine, thus leaving query processing and evaluation to a remote server. This option is possible and makes sense since RDF databases are meant to be accessible through web endpoints and not necessarily as in-memory data.

As stated in Section 2, a shape schema $\mathcal{S}$ is *non-recursive* or *acyclic* if its dependency graph $G_{\mathcal{S}}$ does not have any shape reference cycles, as the one depicted in Figure 3.1. We denote this fragment as $\mathcal{L}^{\text{non-rec}}$, and has already been showed that its validation is tractable in data complexity, in (Corman et al., 2019). The validation problem for acyclic schemas is easy because checking satisfiability of a constraint on any node can be done in a bounded amount of calls. This gives as insight that when constructing schemas tractable evaluation can be guaranteed if the schema does not contain cycle dependencies.

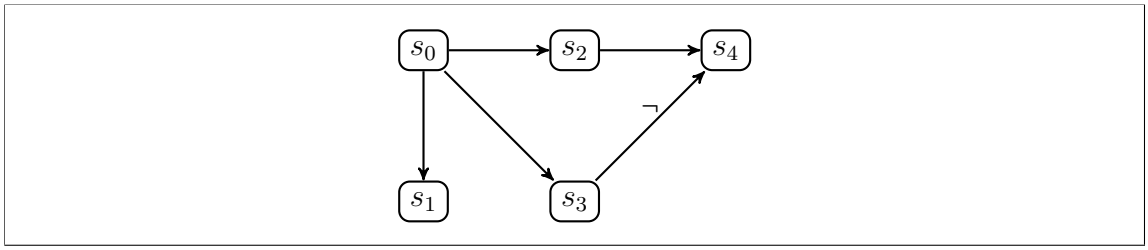

FIGURE 3.1. Example of dependency graph for non-recursive schema.

3.1. In-memory approaches

First, we study two in-memory algorithmic approaches to the validation of acyclic schemas over graph databases. Both of them are based on a specific order for the evaluation of the target nodes present in the schema and show a way to construct a valid assignment, if one exists.

Both methods will consider an acyclic shape schema $\mathcal{S} = \langle S, \mathrm{targ}, \mathrm{def} \rangle$ to be verified over a graph database $\mathcal{G}$ and do so by constructing an assignment σ to check validity of the graph. Since $\mathcal{S}$ is acyclic, a topological order $\geq_{\mathcal{S}}$ can be defined over the different shapes in $\mathcal{S}$ based on the shape reference graph $G_{\mathcal{S}}$. This order assures a that a shape s appears before any shape s' that is referenced in $\mathrm{def}(s)$. In Figure 3.1, nodes are labeled in a topological order, i.e. s_0, s_1, s_2, s_3, s_4 is a valid topological order for that schema. Let $\min_{\geq_{\mathcal{S}}}$ and $\max_{\geq_{\mathcal{S}}}$ be the minimum and maximum shape based on this order $\geq_{\mathcal{S}}$.

Backward Chaining evaluation. This algorithm is based on building a total faithful assignment shape by shape, following the reverse order of $\geq_{\mathcal{S}}$. Starting by $\max_{\geq_{\mathcal{S}}}$, each node in the graph is tested to suffice the corresponding condition $\mathrm{def}(\max_{\geq_{\mathcal{S}}})$. Following the reverse order, verification starts by shapes that do not reference other shapes. Then, when testing constraints that do reference other shapes, shape conditions are verified as simple IRI, since those referenced shapes are already tested. If a target node that does not satisfy its corresponding shape constraint, the algorithm stops and returns a negative answer. A pseudo-implementation is showed in Algorithm 1.

Forward Chaining evaluation. This algorithm also answers the validation problem by building a faithful assignment σ shape by shape but following the actual topological order of shapes $\geq_{\mathcal{S}}$ and only checking for satisfaction of constrains over the nodes needed for validity. By this we mean, starting with $\min_{\geq_{\mathcal{S}}}$, the algorithm checks satisfaction for the corresponding target nodes. This potentially involves checking satisfaction constrains of other shapes over neighbor nodes, and does so recursively until it can answer for the original checked node. It also does so in a way that keeps track of already checked nodes so it does not check satisfaction of a shape over a node twice, and a partial assignment

Algorithm 1 BACKWARD CHAINING EVALUATION ALGORITHM

Require: Schema $\mathcal{S} = \langle S, \text{targ}, \text{def} \rangle$, graph $\mathcal{G}$ and topological order $\geq_{\mathcal{S}}$

$\sigma \leftarrow \varnothing$
for all $s \in S$ in reverse following $\geq_{\mathcal{S}}$ **do**
  $T \leftarrow [\![\text{targ}(s)]\!]^{\mathcal{G}}$
  for all $v \in \mathcal{G}$ **do**
    if $[\text{def}(s)]^{v,\mathcal{G},\sigma} = 1$ **then**
      $\sigma \leftarrow \sigma \cup \{s(v)\}$
    else
      if $v \in T$ **then return** `FALSE`
      end if
      $\sigma \leftarrow \sigma \cup \{\neg s(v)\}$
    end if
  end for
end for
return σ

is built on the fly. A pseudo-implementation is showed in Algorithm 2. The main nodes that are verified are target nodes for each shape, and if at some point any of them does not satisfy its corresponding constraint, then the algorithm stops and returns a negative answer. For simplicity, we omit the implementation of the procedure CHECK that verifies satisfaction of a node v of a constraint formula $\text{def}(s)$ considering the rest of the graph $\mathcal{G}$ and the current assignment σ. CHECK does so in a way that may call itself to check over neighbors of v if another constraint suffices, but always updates the same assignment given σ so that previous validations can be used.

We will state correctness and time execution bounds for both algorithms separately, in the next two propositions. These will be in terms of the size of the set of shape names $|S|$, the size of the graph $|\mathcal{G}|$ considered as the amount of nodes in it ($|V_{\mathcal{G}}|$) and a bound over pattern edges $|E_{\mathcal{G}}|$ named *property maximum* defined as $M_{\mathcal{G}} = \max_{:\mathtt{p} \in \text{IRI}} |\{(v_1, :\mathtt{p}, v_2) \in E_{\mathcal{G}}\}|$. Meaning $M_{\mathcal{G}}$ is the maximum number of times a single edge label appears in the graph. Even though $M_{\mathcal{G}}$ could be similar in magnitude to the whole number of edges in $\mathcal{G}$, general and real graph databases are heterogeneous and have multiple labels, thus the bound generally should be just a small fraction of the total number of edges.

Algorithm 2 FORWARD CHAINING EVALUATION

Require: Schema $\mathcal{S} = \langle S, \text{targ}, \text{def} \rangle$, graph $\mathcal{G}$ and topological order $\geq_{\mathcal{S}}$

$\sigma \leftarrow \varnothing$
for all $s \in S$ in order following $\geq_{\mathcal{S}}$ **do**
  for all $v \in [\![\text{targ}(s)]\!]^{\mathcal{G}}$ **do**
    if $\neg s(v) \in \sigma$ **then return** `FALSE`
    else if $s(v) \notin \sigma$ **then**
      CHECK$(v, \text{def}(s), \sigma, \mathcal{G})$
      if $\neg s(v) \in \sigma$ **then return** `FALSE`
      end if
    end if
  end for
end for
return σ

Proposition 3.1. *The Backward Chaining Algorithm returns a correct answer for the validation problem of a non-recursive shape schema $\mathcal{S} = \langle S, \text{targ}, \text{def} \rangle$ over a graph database $\mathcal{G}$ with property maximum $M_{\mathcal{G}}$, and runs in $\mathcal{O}(M_{\mathcal{G}} \cdot |\mathcal{G}| \cdot |S|)$.*

PROOF. Correctness of the result follows directly from the iterative construction of a valid assignment for each tested constraint $\text{def}(s)$ and from the fact that for acyclic schemas the satisfiability for each shape-node pair can be checked locally. As for running complexity, we will prove it by analyzing different cases of shapes.

Since the Backwards Chaining follows the reverse topological order of shapes in $\mathcal{S}$, all shapes that do not reference other shapes will be evaluated first. If we consider evaluation of those constraints, the defined semantics suggests that they can generally be tested locally over the node. The only exception arises when considering one or more path statements ($\geq_n r.\phi$). These imply navigating thought the graph using the current node as starting point and testing conditions over other nodes. This becomes harder if considering nested path statements, such as: $\geq_2 r.(\geq_3 t.(\geq_1 w.\top))$. In general, the amount of nodes that could be traversed to check the whole constraint grows exponentially. But, since $M_{\mathcal{G}}$ bounds the amount of edges that are equally labeled in the graph $\mathcal{G}$, the amount of nodes at each level of traversal is bounded by $M_{\mathcal{G}}$, since each level share the incoming edge label.

Therefore, in terms of $M_{\mathcal{G}}$, the leaf-traversal nodes are at most $M_{\mathcal{G}}$, and the total node traversed is bounded by $\eta_{\mathcal{S}} \cdot M_{\mathcal{G}}$, where $\eta_{\mathcal{S}}$ is the maximum number of times path statements appear in a single constraint formulas for shapes in $\mathcal{S}$.

Since the procedure follow the reverse order of $\geq_{\mathcal{S}}$ to check constraint formulas, for any shape s that references other shape in $\mathrm{def}(s)$, then the current constructed assignment σ already contains the positive or negative atoms of any shape that could be referenced. Therefore, checking a sub-formula s' can be easily tested checking the membership of $s'(v)$ or $\neg s'(v)$ in the current assignment, instead of recursively testing $\mathrm{def}(s')$. If implemented correctly, this checking takes $\mathcal{O}(1)$. Therefore, since shape-referencing atoms do take constant time to check in this setting, the path statements imply the most working load, but still bounded as the previous argument.

This means, that this strategy takes $\mathcal{O}(\eta_{\mathcal{S}} \cdot M_{\mathcal{G}} \cdot |\mathcal{G}| \cdot |\mathcal{S}|)$ to construct an assignment defined for all nodes in $\mathcal{G}$. What is left is to construct the target sets for each shape and check if are correctly assigned. Since target queries are also expressed through simple monadic queries, each shape target evaluation is bounded by $M_{\mathcal{G}}$. Whereas, checking if target nodes are correctly assigned is considered to take constant time. Therefore it takes a total of $\mathcal{O}(M_{\mathcal{G}} \cdot |\mathcal{S}|)$, which is under $\mathcal{O}(\eta_{\mathcal{S}} \cdot M_{\mathcal{G}} \cdot |\mathcal{G}| \cdot |\mathcal{S}|)$.

If assuming bounded path statement appearance in constraints, then $\mathcal{O}(M_{\mathcal{G}} \cdot |\mathcal{G}| \cdot |\mathcal{S}|)$ is reached. Also is important to consider that $\mathcal{S}$ could be re-written to equivalent shape schema that reduces $\eta_{\mathcal{S}}$, but this increases the size of $|\mathcal{S}|$ by introducing new shapes that replace multiple path statements.

□

Proposition 3.2. *The Forward Chaining Algorithm returns a correct answer for the validation problem of a non-recursive shape schema $\mathcal{S} = \langle S, \mathrm{targ}, \mathrm{def} \rangle$ over a graph database $\mathcal{G}$ with property maximum $M_{\mathcal{G}}$, and runs in $\mathcal{O}(M_{\mathcal{G}}^2 \cdot |S|^2)$.*

PROOF. As the previous algorithm, correctness of the result follows from the construction of a valid assignment for each tested constraint $\mathrm{def}(s)$ and by checking satisfiability for each shape-node pair locally. The order of evaluation of constraints is actually the same as the previous algorithm, just the call of the procedure is what follows the topological order.

As for running complexity, the main difference with Backward Chaining is the fact it does not evaluate for every possible node in the graph that may not be of interest, possibly verifying shapes multiple times over different sets of nodes. The main loop of the algorithm that verifies that target nodes have already been checked takes $\mathcal{O}(M_{\mathcal{G}} \cdot |S|)$, since the amount of nodes that can target statements is bounded by $M_{\mathcal{G}}$. It's the procedure CHECK that amplifies the complexity.

Similarly as stated in proof for Proposition 3.1, most shape constrains can be locally checked assuming constant time, with the exception of path statements that reference other shapes ($\geq_n r.s'$). Both the graph traversal needed to check the path statement, and the recursive calls to check other shapes over neighbors makes the verification harder. Node traversal does not blow up exponentially when considered in terms of $M_{\mathcal{G}}$, similar as in the previous proof, is bounded by $\eta_{\mathcal{S}} \cdot M_{\mathcal{G}}$, where $\eta_{\mathcal{S}}$ is the maximum number of times path statements appear in a single constraint formulas for shapes in $\mathcal{S}$. On top of that, recursive calls are also bounded since $\mathcal{S}$ is acyclic and references are set to end at some point. On the best case, most of these recursive calls are done over nodes that have already been checked, reducing the verification to constant time. Sadly, these calls can be called over disjointed sets of nodes, meaning the whole recursive calling has to be done over potentially $|S|$ times. Therefore a CHECK call is bounded by $\mathcal{O}(M_{\mathcal{G}} \cdot |S|)$, since the depth of the the recursive calls are bounded by $|S|$ and the number of each node traversal is bounded by $M_{\mathcal{G}}$.

This reaches the proposed total complexity of $\mathcal{O}(M_{\mathcal{G}}^2 \cdot |S|^2)$. It is important to note that this considers that no node assignments are reused, and that the depth of recursive calls is

actually the longest path in the dependency graph $G_{\mathcal{S}}$, not actually S, and that successive calls following topological order of shapes decreases the recursive depth by at least one.

□

This shows two basic ways to approach the problem in-memory for the non-recursive case, as well that they present a trade-off in their use. On one hand, Backward Chaining Evaluation is simple in implementation and returns a total assignment, but for larger graph databases it may present unnecessary computation and therefore blow-up in running time. But on the other, Forward Chaining only returns a partial assignment and presents bigger implementation challenges, specifically for the CHECK procedure, but does improve in performance for bigger graph, specially for heterogeneous ones that have a lower edge pattern bound M.

These results are also a reflection over the fact that non-recursive schemas are easy to validate over a graph, and it is the size of the graph its biggest challenge. Both approaches actually do the same verification but in different orders: they check satisfaction of constrains directly for every needed pair of shapes and nodes. This property may not be possible for recursive schemas, as a cycle of satisfaction dependencies can arise.

3.2. Online approach

Now, we analyze an online approach to validation evaluation. Specifically, we address the question of whether the validation problem can be performed by evaluating a single SPARQL query over an endpoint that exposes RDF graph database $\mathcal{G}$. When possible, this approach has a some advantages, as it allows us to validate graphs over endpoints without doing any local computation. On the other hand, if the translation of a schema into SPARQL is too complicated, endpoints may have too much trouble to compute the answer, or it may even be forced to timeout. Thus, one also needs to be able to produce queries that can be evaluated rather efficiently.

We say that a schema $\mathcal{S}$ can be *expressed* in SPARQL if there is a SPARQL query $Q_{\mathcal{S}}$ such that, for every graph $\mathcal{G}$, it holds that $\mathcal{G}$ is valid against $\mathcal{S}$ if and only if $[\![Q_S]\!]^{\mathcal{G}} = \varnothing$. We will show that non-recursive schemas are indeed expressible in SPARQL, and will do so by presenting an actual translation for it.

Given a non-recursive shape schema $\mathcal{S} = \langle S, \mathrm{targ}, \mathrm{def}\rangle$, we define a set of SPARQL queries for each shape: $\mathcal{Q}^{\mathcal{S}} = \{q^s \mid s \in S\}$. Each query q^s is a `SELECT` SPARQL query that validates the faithfulness of that shapes target nodes in a specific RDF graph. It has the following general form:

$q^s :=$ `SELECT ?x`
`WHERE {` $\mathcal{T}(\mathrm{targ}(s), \texttt{?x})$
`FILTER NOT EXISTS {` $\mathcal{C}(\mathrm{def}(s), \texttt{?x})$ `}`
`}`

Where:

- $\mathcal{T}(\text{targ}(s), \texttt{?x})$ is the pattern that matches all targeting nodes as defined by $\text{targ}(s)$ and referenced inside the query with the variable `?x`. Thanks to SPARQL subquery syntax, this could literally translate to $\text{targ}(s)$, but using `?x` as the projection variable.
- $\mathcal{C}(\text{def}(s), \texttt{?x})$ is also a pattern, but defined recursively from $\text{def}(s)$ such that the variable `?x` matches the constraint defined by $\text{def}(s)$.

The proposed recursive conversion of $\mathcal{C}(\text{def}(s), \texttt{?x})$ is defined as follows:

$$
\begin{aligned}
\mathcal{C}(\top, \texttt{?x}) &:= \texttt{adom(?x)} \\
\mathcal{C}(I, \texttt{?x}) &:= \texttt{adom(?x).}\ \ \texttt{FILTER(?x =}\ I\texttt{)} \\
\mathcal{C}(\text{EQ}(r_1, r_2), \texttt{?x}) &:= \texttt{adom(?x).}\ \ \texttt{FILTER NOT EXISTS\{} \\
&\quad \texttt{\{?x}\ \ r_1\ \texttt{?y.}\ \ \texttt{FILTER NOT EXISTS\{ ?x}\ \ r_2\ \ \texttt{?y \} \}} \\
&\quad \texttt{UNION} \\
&\quad \texttt{\{?x}\ \ r_2\ \texttt{?z.}\ \ \texttt{FILTER NOT EXISTS\{ ?x}\ \ r_1\ \ \texttt{?z \} \}} \\
&\quad \texttt{\}} \\
\mathcal{C}(s, \texttt{?x}) &:= \mathcal{C}(\text{def}(s), \texttt{?x}) \\
\mathcal{C}(\phi \wedge \psi, \texttt{?x}) &:= \{\ \mathcal{C}(\phi, \texttt{?x})\ \texttt{.}\ \mathcal{C}(\psi, \texttt{?x})\ \} \\
\mathcal{C}(\neg\phi, \texttt{?x}) &:= \texttt{adom(?x).}\ \ \texttt{FILTER NOT EXISTS\{}\ \mathcal{C}(\phi, \texttt{?x})\ \texttt{\}} \\
\mathcal{C}(\geq_n r.\phi, \texttt{?x}) &:= \texttt{\{ ?x}\ \ r\ \ \texttt{?y1, ?y2, ..., ?yn.} \\
&\quad \mathcal{C}(\phi, \texttt{?y1})\ \texttt{.} \\
&\quad \mathcal{C}(\phi, \texttt{?y2})\texttt{.} \\
&\quad \texttt{...} \\
&\quad \mathcal{C}(\phi, \texttt{?yn})\ \texttt{.} \\
&\quad \texttt{FILTER(?y1 != ?y2) .} \\
&\quad \texttt{...} \\
&\quad \texttt{FILTER(?y1 != ?yn) .} \\
&\quad \texttt{...} \\
&\quad \texttt{\}}
\end{aligned}
$$

Where `adom(?x)` = { `?x ?x1 ?x2` } `UNION` { `?x2 ?x ?x4` } `UNION` { `?x5 ?x6 ?x` }. These translations apply for the general non-recursive case, but many combinations can be further optimized in size of translation. The `adom` atom particularly can be omitted in most translations if the variable is already declared on an outer scope. Also, every time a new SPARQL variable, as `?y1`, is introduced, it should be a variable that has not been used before in the rest of the translation. This assures there is no incorrect referencing in different scopes.

Is easy to see that the given translation for any finite constraint $\mathrm{def}(s)$ that does not mention any shape name in it will directly result into a finite pattern. But, if we introduce the use of referencing other shape names, the translation may not terminate as the translation may get stuck applying the rule: $\mathcal{C}(s, \texttt{?x}) \coloneqq \mathcal{C}(\mathrm{def}(s), \texttt{?x})$. This is true in the general case, but not for the non-recursive schemas, as showed by the following lemma:

Lemma 3.1. *A non-recursive shape schema* $\mathcal{S} = \langle S, \mathrm{targ}, \mathrm{def}\rangle$ *can be successfully converted to a set of valid* SPARQL *queries* $\mathcal{Q}^{\mathcal{S}}$.

PROOF. As stated before, $\mathcal{Q}^{\mathcal{S}} = \{\ q^s \mid s \in S\}$, therefore, is suffices to show that each individual q^s is a valid SPARQL `SELECT` query.

Let s be an arbitrary shape name in S. First, note that every constrain $\mathrm{def}(s)$ is finite and applying non-shape rules of translation always resolve in a finite valid pattern. Then, if $\mathcal{C}(\mathrm{def}(s), \texttt{?x})$ does not resolve into a valid pattern using the previously described recursive translation, then another shape name $s' \in S$ must appear in $\mathrm{def}(s)$ such that the translation continues. Furthermore, the same argument can be applied over s', for the existence of another shape s''. By applying this argument $|S|$ times, a sequence of shape names $s_1, s_2, \ldots, s_{|S|+1}$ can be defined such that each s_{i+1} appears in $\mathrm{def}(s_i)$. By pigeonhole principle, two elements in the sequence must be the same shape, and therefore, a cycle exists in the sequence. This directly means that a cycle must exists in the dependency graph$G_{\mathcal{S}}$, which is a contradiction. Therefore our initial assumption that $\mathcal{C}(\mathrm{def}(s), \texttt{?x})$ does not resolve into a valid pattern must be false.

Consequently, for any shape $s \in S$, Q^s is a valid SPARQL `SELECT` query, and $\mathcal{Q}^{\mathcal{S}}$ is a valid set of SPARQL queries.

□

The rest of this section is spent proving that the translation is indeed correct and make non-recursive schemas expressible in SPARQL.

Proposition 3.3. *Every schema in $\mathcal{L}^{\text{non-rec}}$ can be expressed in* SPARQL.

PROOF. For a given shape schema $\mathcal{S}$ and its SPARQL translation set $\mathcal{Q}^{\mathcal{S}}$, the SPARQL query Q defined as the query that returns the union of results from every single shape query $q^s \in \mathcal{Q}^{\mathcal{S}}$ is a single query that can determine the validity of a graph $\mathcal{G}$:

$$:= \texttt{SELECT ?x WHERE } \{q^{s_1} \texttt{ UNION } q^{s_2} \texttt{ UNION ... }\}$$

Then, for every non-recursive shape schema $\mathcal{S}$, the mentioned SPARQL query Q can be computed such that an RDF graph $\mathcal{G}$ is valid against $\mathcal{S}$ if and only if $[\![Q]\!]^{\mathcal{G}} = \varnothing$. We will prove both directions of correctness separately, but nonetheless both assume that shape schema $\mathcal{S}$ is in a certain normal form, *simple shape normal form*. $\mathcal{S}$ is in simple shape normal form if for every shape $s \in S$, its corresponding constraint $\mathrm{def}(s)$ belong to the language of:

$$\phi \ ::= \ \top \mid s \mid \neg s \mid I \mid s_1 \wedge s_2 \mid \ \geq_n r.s \mid \mathrm{EQ}(r_1, r_2)$$

Where $I \in IRI$, $s, s_1, s_2 \in S$, and r, r_1, r_2 are edge patterns. If a non-recursive schema $\mathcal{S}$ is not in simple shape normal form, it can be easily converted by extending the shape name set with new shapes with no target queries and replacing all non-atomic formulas with these shapes. This ensures all conjunctions and negations are a separate shape in the schema.

First, we will prove the right direction of correctness, i.e. if $\mathcal{G}$ is valid against $\mathcal{S}$ then $[\![Q]\!]^{\mathcal{G}} = \varnothing$. If $\mathcal{G} = \langle V_{\mathcal{G}}, E_{\mathcal{G}} \rangle$ is valid against $\mathcal{S} = \langle S, \mathrm{targ}, \mathrm{def} \rangle$, then a faithful assignment for

$\mathcal{G}$ and $\mathcal{S}$ must exist. Furthermore, since a non-recursive schema is also stratified, a faithful and total assignment σ must exist, as showed in (Corman et al., 2018b) (Proposition 3).

The `SELECT` SPARQL queries semantics dictate that the result of evaluating such query will be $\varnothing$ if no triple can be found that suffices the given pattern in the `WHERE` clause. Since Q is a union of different sub-queries q^s, an empty result means the same result in every single sub-query. The general form for query q^s assures that all target nodes are matched, but are filtered out by the `NOT EXISTS` clause. Therefore, in order to show that $[\![Q^s]\!]^{\mathcal{G}} = \varnothing$ holds for every $s \in S$, it suffices to show that for every variable mapping $\mu \in [\![\mathcal{T}(\mathrm{targ}(s), \texttt{?x})]\!]^{\mathcal{G}}$, it also exists another variable mapping $\mu' \in [\![\mathcal{C}(\mathrm{def}(s), \texttt{?x})]\!]^{\mathcal{G}}$ such that $\mu(\texttt{?x}) = \mu'(\texttt{?x})$. That way, every target node is then filtered out thanks to the `NOT EXISTS` clause, and therefore $\varnothing$ is obtained.

By structural induction over the simple shape normal form grammar, it can be showed that given the total and faithful assignment σ and shape constraint $\mathrm{def}(s)$, the assignment is consistent with the constraint pattern in the `NOT EXISTS` clause of q^s, i.e. :

$$[\mathrm{def}(s)]^{\mathcal{G},v,\sigma} = 1 \iff \exists \mu \in [\![\mathcal{C}(\mathrm{def}(s), \texttt{?x})]\!]^{\mathcal{G}}.\ \mu(\texttt{?x}) = v$$

First, the base cases:

- $\mathrm{def}(s) = \top$

$$[\![\mathcal{C}(\top, \texttt{?x})]\!]^{\mathcal{G}} = [\![\texttt{adom(?x)}]\!]^{\mathcal{G}} = \{(\texttt{?x} \to u) \mid u \in V_{\mathcal{G}}\}$$

  For any node v in $\mathcal{G}$, the mapping μ that assigns `?x` to v must be in $[\![\mathcal{C}(\mathrm{def}(s), \texttt{?x})]\!]^{\mathcal{G}}$, since it exists for any node in $V_{\mathcal{G}}$. And, by $\top$ semantics, $[\top]^{\mathcal{G},v,\sigma} = 1$ always holds.

- $\mathrm{def}(s) = I$

$$[\![\mathcal{C}(I, \texttt{?x})]\!]^{\mathcal{G}} = [\![\texttt{adom(?x) FILTER(I = ?x)}]\!]^{\mathcal{G}}$$
$$= \{(\texttt{?x} \to u) \mid u = I\}$$

$[I]^{\mathcal{G},v,\sigma} = 1$, if and only if $v = I$, which is equivalent to the existence of the mapping μ that assigns `?x` to v in $[\![\mathcal{C}(I, \texttt{?x})]\!]^{\mathcal{G}}$.

- $\mathrm{def}(s) = \mathrm{EQ}(r_1, r_2)$

$$
\begin{aligned}
[\![\mathcal{C}(\mathrm{EQ}(r_1, r_2), \texttt{?x})]\!]^{\mathcal{G}} = [\![& \texttt{adom(?x).} \quad \texttt{FILTER NOT EXISTS\{} \\
& \quad \texttt{\{?x } r_1 \texttt{ ?y.} \quad \texttt{FILTER NOT EXISTS\{ ?x } r_2 \texttt{ ?y \} \}} \\
& \texttt{UNION} \\
& \quad \texttt{\{?x } r_2 \texttt{ ?z.} \quad \texttt{FILTER NOT EXISTS\{ ?x } r_1 \texttt{ ?z \} \}} \\
& \texttt{\}}]\!]^{\mathcal{G}} \\
= \{ & (\texttt{?x} \to u) \mid \forall u' \in V_{\mathcal{G}}. \text{ where} \\
& (u, u') \in r_1(\mathcal{G}) \leftrightarrow (u, u') \in r_2(\mathcal{G})\}
\end{aligned}
$$

$[\mathrm{EQ}(r_1, r_2)]^{\mathcal{G},v,\sigma} = 1$ if and only if $\{v' \mid (v, v') \in r_1(\mathcal{G})\} = \{v' \mid (v, v') \in r_2(\mathcal{G})\}$. The latter is equivalent to the property that there are no connected nodes to v only through an r_1-labeled or r_2-labeled edges. This is what $\mathcal{C}(\mathrm{EQ}(r_1, r_2), \texttt{?x})$ captures, and therefore mapping μ such that $\mu(\texttt{?x}) = v$ is in $[\![\mathcal{C}(\mathrm{EQ}(r_1, r_2), \texttt{?x})]\!]^{\mathcal{G}}$.

Given the base cases, we proceed to prove the induction step for all recursive cases:

- $\mathrm{def}(s) = s'$

$$[\![\mathcal{C}(s', \texttt{?x})]\!]^{G} = [\![\mathcal{C}(\mathrm{def}(s'), \texttt{?x})]\!]^{G}$$

If $[s']^{\mathcal{G},v,\sigma} = 1$ then by semantics, $s'(v) \in \sigma$. Since σ is faithful $[\mathrm{def}(s')]^{\mathcal{G},v,\sigma} = 1$ holds. Thanks to the induction hypothesis since $[\mathrm{def}(s')]^{\mathcal{G},v,\sigma} = 1$, then a mapping μ must exists in $[\![\mathcal{C}(\mathrm{def}(s'), \texttt{?x})]\!]^{\mathcal{G}}$, which is equivalent to $[\![\mathcal{C}(s', \texttt{?x})]\!]^{\mathcal{G}}$. For the other direction, if $\exists \mu \in [\![\mathcal{C}(s', \texttt{?x})]\!]^{\mathcal{G}}$ where $\mu(\texttt{?x}) = v$, then it also holds that $\mu \in [\![\mathcal{C}(\mathrm{def}(s'), \texttt{?x})]\!]^{\mathcal{G}}$. By

induction hypothesis, $[\mathrm{def}(s')]^{\mathcal{G},v,\sigma} = 1$ then holds. Since σ is total and faithful, then $s'(v) \in \sigma$, and therefore $[s']^{\mathcal{G},v,\sigma} = 1$.

- $\mathrm{def}(s) = \neg s'$

$$[\![\mathcal{C}(\neg s', ?\mathtt{x})]\!]^G = [\![\mathtt{adom(?x)\ .\ \ FILTER\ NOT\ EXISTS}\{\mathcal{C}(s', ?\mathtt{x})\}]\!]^G$$
$$= \{(?\mathtt{x} \to u) \mid \nexists\, \mu \in [\![\mathcal{C}(s', ?\mathtt{x})]\!]^G.\ \mu(?\mathtt{x}) = u\}$$

$[\neg s']^{\mathcal{G},v,\sigma} = 1$ if and only if $[s']^{\mathcal{G},v,\sigma} = 0$. Using the contrapositive of our hypothesis, $[s']^{\mathcal{G},v,\sigma} = 0$ indicates that there is no mapping in $[\![\mathcal{C}(s', ?\mathtt{x})]\!]^{\mathcal{G}}$ such that $\mu(?\mathtt{x}) = v$, and vice versa. Therefore, the SPARQL pattern evaluation indicates a mapping $\mu \in [\![\mathcal{C}(\neg s', ?\mathtt{x})]\!]^{\mathcal{G}}$ must exist for v.

- $\mathrm{def}(s) = s_1 \wedge s_2$

$$[\![\mathcal{C}(s_1 \wedge s_2, ?\mathtt{x})]\!]^{\mathcal{G}} = [\![\ \mathcal{C}(s_1, ?\mathtt{x})\ \ .\ \ \mathcal{C}(s_2, ?\mathtt{x})\ ]\!]^{\mathcal{G}}$$
$$= \{\mu \mid \exists \mu_1, \mu_2.$$
$$\mu_1 \in [\![\mathcal{C}(s_1, ?\mathtt{x})]\!]^{\mathcal{G}},$$
$$\mu_2 \in [\![\mathcal{C}(s_2, ?\mathtt{x})]\!]^{\mathcal{G}}.$$
$$\mu_1(?\mathtt{x}) = \mu_2(?\mathtt{x}) \ \wedge\ \mu = \mu_1 \cup \mu_2\}$$

$[s_1 \wedge s_2]^{\mathcal{G},v,\sigma} = 1$ holds if and only if both $[s_1]^{\mathcal{G},v,\sigma} = 1$ and $[s_2]^{\mathcal{G},v,\sigma} = 1$ hold. By induction hypothesis, $\exists \mu_1, \mu_2$ such that $\mu_1(x) = v,\ \mu_1 \in [\![\mathcal{C}(s_1, ?\mathtt{x})]\!]^{\mathcal{G}}$ and $\mu_2(x) = v,\ \mu_2 \in [\![\mathcal{C}(s_2, ?\mathtt{x})]\!]^{\mathcal{G}}$. Since it can be assumed that the intersection of variables between μ_1 and μ_2 is only $?\mathtt{x}$, and they assign $?\mathtt{x}$ to the same node v, then they are compatible and can be joined without inconsistencies into $\mu = \mu_1 \cup \mu_2$. Note that for that same reason, μ is equivalent to both when considering the corresponding pattern evaluations, and therefore, $\mu \in [\![\ \mathcal{C}(s_1, ?\mathtt{x})\ \ .\ \ \mathcal{C}(s_2, ?\mathtt{x})\ ]\!]^{\mathcal{G}}$.

- $\mathrm{def}(s) = \geq_n r.s'$

$$
\begin{aligned}
[\![\mathcal{C}(\geq_n r.s', \texttt{?x})]\!]^{\mathcal{G}} = [\![\{\ & \texttt{?x}\ r\ \texttt{?y1, ?y2, ..., ?yn.} \\
& \mathcal{C}(s', \texttt{?y1}). \\
& \mathcal{C}(s', \texttt{?y2}). \\
& \ldots \\
& \mathcal{C}(s', \texttt{?yn}). \\
& \texttt{FILTER(?y1 != ?y2) .} \\
& \ldots \\
& \texttt{FILTER(?y1 != ?yn) .} \\
& \ldots \\
& \}\ ]\!]^{\mathcal{G}} \\
= \{\mu \mid\ & \exists \mu_1 \in [\![\mathcal{C}(s', \texttt{?y1})]\!]^{\mathcal{G}},\ \exists \mu_2 \in [\![\mathcal{C}(s', \texttt{?y2})]\!]^{\mathcal{G}}, \\
& \ldots,\ \exists \mu_n \in [\![\mathcal{C}(s', \texttt{?yn})]\!]^{\mathcal{G}}. \\
& \text{such that } \forall i:\ (u, \mu_i(\texttt{?yi})) \in r(\mathcal{G}), \\
& |\{\mu_1(\texttt{?y1}), \mu_2(\texttt{?y2}), \ldots, \mu_n(\texttt{?yn})\}| = n, \\
& \text{and } \mu = \mu_1 \cup \mu_2 \cup \ldots \cup \mu_n \cup (\texttt{?x} \to u)\}
\end{aligned}
$$

$[\geq_n r.s']^{\mathcal{G},v,\sigma} = 1$ holds if and only if $\exists v_1, v_2, \ldots, v_n$ such that are all different nodes, $(v, v_i) \in r(\mathcal{G})$ and $[s']^{\mathcal{G},v_i,\sigma} = 1$ for all $1 \leq i \leq n$. Therefore, thanks to induction hypothesis $\exists \mu_i \in [\![\mathcal{C}(s', \texttt{?yi})]\!]^{\mathcal{G}}$ such that $\mu_i(\texttt{?yi}) = v_i$, for all i. Since `x` does not appear in any pattern $\mathcal{C}(s', \texttt{?yi})$ and no pattern share variables, $\mu_1, \mu_2, \ldots, \mu_n$ can easily be joined into μ and it can be extended by assigning $\mu(\texttt{x}) = v$. Thus, $\mu \in [\![\mathcal{C}(\geq_n r.s', \texttt{?x})]\!]^{\mathcal{G}}$.

This completes all inductive cases, thus proving that for a total and faithful assignment σ and formula $\mathrm{def}(s)$, σ is consistent with the constraint pattern in the `NOT EXISTS` clause of q^s.

Now, let $s \in S$ be any shape name in the schema and q^s its corresponding query. Consider any node a from $\mathcal{G}$ such that for a mapping $\mu \in [\![\mathcal{T}(\mathrm{targ}(s), ?\mathtt{x})]\!]^{\mathcal{G}}$ and $\mu(?\mathtt{x}) = a$. This means that $a \in [\![\mathrm{targ}(s)]\!]^{\mathcal{G}}$. Since σ is faithful, $s \in \sigma(a)$ and $[\mathrm{def}(s)]^{\mathcal{G},a,\sigma} = 1$. The previous induction showed that a mapping μ' exists such that $\mu'(?\mathtt{x}) = a$ and $\mu' \in [\![\mathcal{C}(\mathrm{def}(s), ?\mathtt{x})]\!]^{\mathcal{G}}$. Then, for both arbitrary s and a, a mapping that assigns a to suffice $\mathcal{C}(\mathrm{def}(s), ?\mathtt{x})$ exists. Then all target nodes of s are not contained in $[\![Q^s]\!]^{\mathcal{G}}$, and therefore, $[\![q^s]\!]^{\mathcal{G}} = \varnothing$ for every shape s. Thus, the final result of the whole query Q on $\mathcal{G}$ is also: $[\![Q]\!]^{\mathcal{G}} = \varnothing$.

The opposite direction states that if $[\![Q]\!]^{\mathcal{G}} = \varnothing$ then $\mathcal{G}$ is valid against $\mathcal{S}$. A faithful assignment σ must be found for $\mathcal{G}$ and $\mathcal{S}$, given that $[\![Q]\!]^{\mathcal{G}} = \varnothing$. The latter implies that $\forall q^s \in \mathcal{Q}^S,\ [\![q^s]\!]^{\mathcal{G}} = \varnothing$. Consider the assignment σ defined as follows for any $v \in \mathcal{G}$ and $s \in S$:

$$s(v) \in \sigma \longleftrightarrow \exists \mu \in [\![\mathcal{C}(\mathrm{def}(s), ?\mathtt{x})]\!]^{\mathcal{G}}.\ \mu(?\mathtt{x}) = v$$

$$\neg s(v) \in \sigma \longleftrightarrow \nexists\ \mu \in [\![\mathcal{C}(\mathrm{def}(s), ?\mathtt{x})]\!]^{\mathcal{G}}.\ \mu(?\mathtt{x}) = v$$

What is left to prove, is that σ is indeed faithful for graph $\mathcal{G} = \langle V_{\mathcal{G}}, E_{\mathcal{G}} \rangle$ and schema $\mathcal{S} = \langle S, \mathrm{targ}, \mathrm{def} \rangle$.

Given an arbitrary shape name $s \in S$, and node $v \in [\![\mathrm{targ}(s)]\!]^{\mathcal{G}}$, by definition a mapping must exists $\mu \in [\![\mathcal{T}(\mathrm{targ}(s), ?\mathtt{x})]\!]^{\mathcal{G}}$ such that $\mu(\mathtt{x}) = v$. Since $[\![Q^s]\!]^{\mathcal{G}} = \varnothing$, there must exists $\mu' \in [\![\mathcal{C}(\mathrm{def}(s), ?\mathtt{x})]\!]^{\mathcal{G}}.\ \mu'(\mathtt{x}) = v$ and therefore, $s(v) \in \sigma$. Then all target nodes are correctly s-labeled: $[\![\mathrm{targ}(s)]\!]^{\mathcal{G}} \subseteq \{v \mid s(v) \in \sigma\}$. Since the shape was also arbitrary, it must hold for all shape names in S.

Now, given an arbitrary shape $s \in S$ and node $v \in \mathcal{G}$, the following should hold for a faithful assignment: if $s(v) \in \sigma$, then $[\mathrm{def}(s)]^{\mathcal{G},v,\sigma} = 1$, and if $\neg s(v) \in \sigma$ then $[\mathrm{def}(s)]^{\mathcal{G},v,\sigma} = 0$.

It can also be shown by structural induction over the simple shape normal form grammar for $\mathrm{def}(s)$ that the latter holds. First, the base cases:

- $\mathrm{def}(s) = \top$:

  If $s(v) \in \sigma$, then a mapping that matches to v exists in $[\![\mathcal{C}(\mathrm{def}(\top), ?\mathtt{x})]\!]^{\mathcal{G}}$. The existence of μ validates v as a node in $\mathcal{G}$, and as such, it should also hold that $[\top]^{v,G,\sigma} = 1$.

  On the other hand, if $\neg s(v) \in \sigma$, since $[\![\mathcal{C}(\top, ?\mathtt{x})]\!]^{\mathcal{G}}$ has mappings matching every node in $\mathcal{G}$, a contradiction is reached since it would mean v is not a valid node in $\mathcal{G}$ or that there are no nodes in $\mathcal{G}$. Then this base case can not occur for a not empty graph $\mathcal{G}$.
- $\mathrm{def}(s) = I$:

  If $s(v) \in \sigma$, then a mapping that matches to v exists in $[\![\mathcal{C}(\mathrm{def}(I), ?\mathtt{x})]\!]^{\mathcal{G}}$, which matches only equivalent nodes to I. Therefore $v = I$ and $[I]^{v,G,\sigma} = 1$.

  Whereas, if $\neg s(v) \in \sigma$, then no mappings match to v, meaning $v \neq I$, and therefore $[I]^{v,G,\sigma} = 0$.
- $\mathrm{def}(s) = \mathrm{EQ}(r_1, r_2)$:

  If $s(v) \in \sigma$, then a mapping μ that matches to v exists in $[\![\mathcal{C}(\mathrm{def}(\mathrm{EQ}(r_1, r_2)), ?\mathtt{x})]\!]^{\mathcal{G}}$. As showed in the previous direction proof, μ assigns only to nodes such that has no neighbors that are only connected through r_1 or r_2, all of such neighbors are connected with both labeled-edges. Therefore, $[\mathrm{EQ}(r_1, r_2)]^{v,G,\sigma} = 1$.

  If $\neg s(v) \in \sigma$, then there is no mapping matching v in the evaluation of the SPARQL pattern for $\mathcal{C}(\mathrm{EQ}(r_1, r_2), ?\mathtt{x})$, then the must exists at least one neighbor of v that it is only connected through a r_1-labeled edge or a r_2-labeled edge. Therefore it holds that $[\mathrm{EQ}(r_1, r_2)]^{v,G,\sigma} = 0$.

Now, the inductive steps:

- $\mathrm{def}(s) = s'$:

  If $s(v) \in \sigma$, then a mapping μ that matches to v exists in $[\![\mathcal{C}(s', ?\mathtt{x})]\!]^{\mathcal{G}}$. The mapping μ is also in the evaluation of the SPARQL pattern for the formula for s': $\mu \in [\![\mathcal{C}(\mathrm{def}(s'), ?\mathtt{x})]\!]^{\mathcal{G}}$. By the definition of assignment σ, $s'(v) \in \sigma$ holds. Therefore $[s']^{v,G,\sigma} = 1$.

  If $\neg s(v) \in \sigma$, then no mapping exists in $[\![\mathcal{C}(\mathrm{def}\, s', ?\mathtt{x})]\!]^{\mathcal{G}}$ such that $\mu(?\mathtt{x}) = v$. By the definition of σ, $\neg s'(v) \in \sigma$. And therefore $[s']^{v,G,\sigma} = 0$.

- $\mathrm{def}(s) = \neg s'$:

  If $s(v) \in \sigma$, then a mapping μ that matches to v exists in $[\![\mathcal{C}(\neg s', ?\mathtt{x})]\!]^{\mathcal{G}}$. The existence of μ confirms the fact that no mappings matching v exists in $[\![\mathcal{C}(s', ?\mathtt{x})]\!]^{\mathcal{G}}$. Then, $\neg s'(v) \in \sigma$, and by induction hypothesis, $[s']^{v,G,\sigma} = 0$. Therefore $[\neg s']^{v,G,\sigma} = 1$.

  If $\neg s(v) \in \sigma$, then no mapping exists in $[\![\mathcal{C}(\neg s', ?\mathtt{x})]\!]^{\mathcal{G}}$ such that $\mu(?\mathtt{x}) = v$. Since the first part of $\mathcal{C}(\neg s', ?\mathtt{x})$ matches all nodes in $\mathcal{G}$, then a mapping μ matching v must exists that is in $[\![\mathcal{C}(s', ?\mathtt{x})]\!]^{\mathcal{G}}$. Therefore, $s'(v) \in \sigma$ and by induction hypothesis $[s']^{v,G,\sigma} = 1$. This implies $[\neg s']^{v,G,\sigma} = 0$.

- $\mathrm{def}(s) = s_1 \wedge s_2$:

  If $s(v) \in \sigma$, then a mapping μ that matches to v exists in $[\![\mathcal{C}(s_1 \wedge s_2, ?\mathtt{x})]\!]^{\mathcal{G}}$. Mappings μ_1 and μ_2 must exists such that $\mu_1 \in [\![\mathcal{C}(s_1, ?\mathtt{x})]\!]^{\mathcal{G}}$, $\mu_2 \in [\![\mathcal{C}(s_2, ?\mathtt{x})]\!]^{\mathcal{G}}$ and $\mu_1(?\mathtt{x}) = \mu_2(?\mathtt{x}) = v$. By induction hypothesis, $[s_1]^{v,G,\sigma} = 1$ and $[s_2]^{v,G,\sigma} = 1$ and therefore $[s_1 \wedge s_2]^{v,G,\sigma} = 1$.

  If $\neg s(v) \in \sigma$, then no mapping exists in $[\![\mathcal{C}(s_1 \wedge s_2, ?\mathtt{x})]\!]^{\mathcal{G}}$ such that $\mu(?\mathtt{x}) = v$. Since $[\![\mathcal{C}(s_1, ?\mathtt{x})]\!]^{\mathcal{G}}$ and $[\![\mathcal{C}(s_2, ?\mathtt{x})]\!]^{\mathcal{G}}$ do not share variables except for $?\mathtt{x}$, then either one of the mappings sets must be empty (if intersected with mappings that match to v). If that is the case, then by induction hypothesis either $[s_1]^{v,G,\sigma} = 0$ or $[s_2]^{v,G,\sigma} = 0$ hold, and therefore $[s_1 \wedge s_2]^{v,G,\sigma} = 0$.

- $\text{def}(s) = \geq_n r.s'$:

  If $s(v) \in \sigma$, then a mapping μ that matches to v exists in $[\![\mathcal{C}(\geq_n r.s', \texttt{?x})]\!]^{\mathcal{G}}$. Different nodes $v_1, \ldots, v_n$ from $\mathcal{G}$ and mappings $\mu_1, \ldots, \mu_n$ must exists such that $\mu_i \in [\![\mathcal{C}(s', \texttt{?yi})]\!]^{\mathcal{G}}$, $\mu_i(\texttt{?yi}) = v_i$ and $(v, v_i) \in r(\mathcal{G})$, for all i. By induction hypothesis, $[s']^{v_i,G,\sigma} = 1$, and therefore, $[\geq_n r.s']^{v,G,\sigma} = 1$.

  If $\neg s(v) \in \sigma$, then no mapping μ exists in the evaluation of the corresponding pattern such that $\mu(\texttt{?x}) = v$, then there must be strictly less that n different nodes v_i such that $(v, v_i) \in r(G)$ and corresponding mappings such that μ_i in $[\![\mathcal{C}(s', \texttt{?yi})]\!]^{\mathcal{G}}$ where $\mu_i(\texttt{yi}) = v_i$. Therefore, $[\geq_n r.s']^{v,G,\sigma} = 0$.

Thus, proving that the defined assignment σ is indeed faithful, guaranteed by the result of the SPARQL query Q.

Finally, this proves that if $\mathcal{G}$ is valid against $\mathcal{S}$ if and only if $[\![Q]\!]^{\mathcal{G}} = \varnothing$. Therefore, for any given graph $\mathcal{G}$ and non-recursive shape schema $\mathcal{S}$, the set of SPARQL queries $\mathcal{Q}^{\mathcal{S}}$ can be constructed to build the single SPARQL query Q. The latter can then be evaluated over $\mathcal{G}$ and if and only the results are $\varnothing$, then $\mathcal{G}$ is indeed valid against $\mathcal{S}$. Therefore any arbitrary non-recursive schema $\mathcal{S}$ is indeed expressible in SPARQL.

□

This section shows that not only in-memory approaches are in the realm of possibilities when considering validation of SHACL schemas. A whole fragment of schemas can be translated to SPARQL and leave the computation problem to an endpoint instead of the working machine. Sadly, it has been shown in (Corman et al., 2019) that multiple SHACL fragments apart from the non-recursive case cannot be expressed in SPARQL. This will be further described in the next section.

4. VALIDATION FOR RECURSIVE SHACL

In this section, we discuss the rest SHACL schemas, i.e. recursive schemas. Since it was already showed in (Corman et al., 2018a), that validation of full SHACL in data complexity is NP-hard, we revise the known recursive fragments and identify their tractability in order to introduce in the two following sections a new proposed tractable fragment.

Straight away, the difference with the previous discussed fragment is the presence of cycles in the dependency graph of an schema. This detail makes it impossible for approaches like the Backward and Forward Chaining evaluations to work, because checking that a node satisfies a shapes constraint is not necessarily bounded to end. What happens then is that checking satisfaction for a certain node and shape *could* be decided directly with local information of the node, or dependencies between shape assignments are established in a cyclical manner which leave unclear results. For example, a node v could be assigned the shape s only if neighbor node v' is assigned shape s', while the assignment of s' for v' depends on another assignment that depends on the original $s(v)$. The complexity between these generated cyclical dependencies grows with the complexity of the schema.

Separately, Section 2 hinted that the presence of negation in dependency cycles make the validation problem even harder as it introduced the need to differentiate between validation by total and partial assignments. It was also mentioned the categorization of *stratified schemas* as the ones whose dependency graph do not contain a cycle with at least one negation. In (Corman et al., 2018a) they are further described as schemas where a well behaved shape stratum can be defined via a labeling function over the set of shapes of the schema, which cannot be accomplished with negative cycles. For instance, Figure 4.1 shows an example of a recursive and stratified schema. A well behaved stratum can be selected by assigning s_2 the lowest level, and then all the rest a higher one to respect the presence of a negation.

For the rest of this thesis, we will denote the fragment of stratified SHACL as $\mathcal{L}^{\text{strat}}$. Even tough it was shown that for $\mathcal{L}^{\text{strat}}$ the validation problems through total and partial assignments coincide, their general validation problem is still NP-hard in data-complexity.

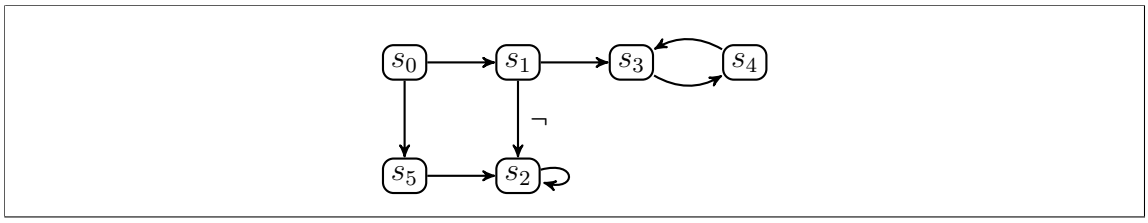

FIGURE 4.1. Example of a stratified schema.

The main issue that arises, is the fact that even through negative cycles are not allowed, multiple positive and negative paths still can arrive to a cycle. This happens in Figure 4.1 between the shapes s_0 and s_2. Then, in validation processing, assignments of the shape s_2 cannot be done independently without cheeking assignments for s_0, because one path may depend on only positive assignments, while the other translates into dependencies for their negation.

On the bright side, there are two recursive but tractable schema fragments already identified: $\mathcal{L}^{+}_{\vee}$ (introduced in (Corman et al., 2018b)) and $\mathcal{L}^{s}$ (introduced in (Corman, Reutter, & Savkovic, 2018c)). The first fragment, $\mathcal{L}^{+}_{\vee}$, we call *positive* SHACL. It is defined by simply disallowing negation to be used in constraints, but it does allow the use of disjunction ($\vee$) as a native operator. Whereas $\mathcal{L}^{s}$, called *strictly stratified* SHACL, allows the use of negation, but restricts the interplay between recursion and negation, even further than $\mathcal{L}^{\text{strat}}$. The left dependency graph in Figure 4.2 corresponds to a positive schema, while the right one corresponds to a strictly stratified schema. As one may guess, both fragments are contained in $\mathcal{L}^{\text{strat}}$, therefore neither allow the use of negation inside a dependency cycle between shapes.

These tractable fragments do achieve the property that stratified SHACL did not, where the assignment of cyclic shapes can be done independently, and the assignment of shapes that depend on them would not alter this decision.

As we can see, negation in the scope of a cycle is not the only source of intractability, the interplay between recursion and negation in general arises problems. It turns out that this interplay also arises between negation and shapes with target queries. Specifically, shapes with negations between them, where both of them have targets nodes to be

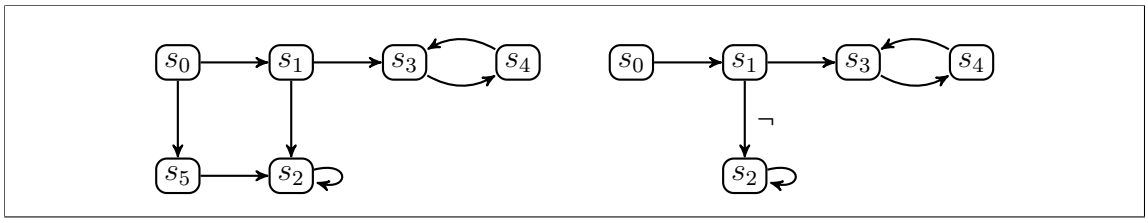

FIGURE 4.2. Examples of positive and strictly stratified schemas.

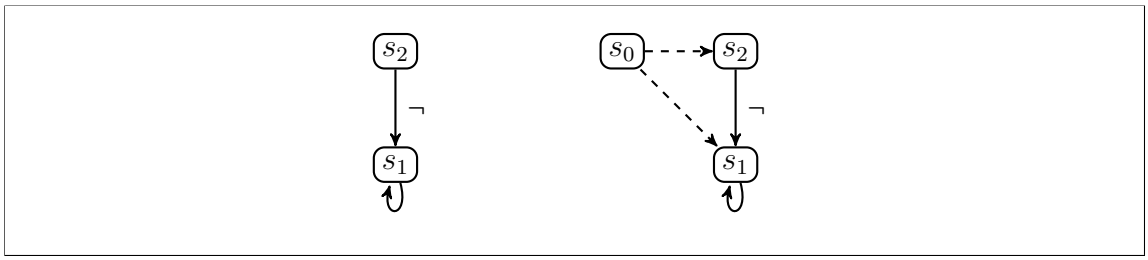

FIGURE 4.3. Simple recursive schema for which validation can become intractable.

validated, in the presence of all positive cycles, can also produce intractability. The left of Figure 4.3 shows the dependency graph of a simple schema of two shapes s_1 and s_2, where s_2 depends on the negation of s_1, while the latter has a loop dependency. If both of them have target queries, then the validation problem can become intractable. Proposition 4.1 proves it as a demonstration by establishing a reduction from the 3CNF-SAT to shape validation using a schema with an equivalent dependency graph.

Proposition 4.1. *A recursive schema $\mathcal{S}$ exists such that 3CNF-SAT is reducible to validation of $\mathcal{S}$ and there are no negative cycles in $G_{\mathcal{S}}$.*

PROOF. 3CNF-SAT takes a propositional formula φ in 3CNF form and then looks to answer if said formula can be satisfied. Now, consider the schema $\mathcal{S}_{\text{3CNF-SAT}} = \langle S, \text{targ}, \text{def} \rangle$ defined as follows:

$$
\begin{aligned}
S = \quad & \{c,\ n\} \\
\text{targ}(c) = \quad & \texttt{SELECT ?x WHERE\{ ?x :type :cla \}} \\
\text{def}(c) = \quad & \geq_1 \texttt{:l}.\ \neg n \\
\text{targ}(n) = \quad & \texttt{SELECT ?x WHERE\{ ?x :type :var \}} \\
\text{def}(n) = \quad & \geq_1 \texttt{:r}.\ n
\end{aligned}
$$

Note that the dependency graph for $\mathcal{S}_{\text{3CNF-SAT}}$ coincides with the left example in Figure 4.3. To show that 3CNF-SAT can be reduced to validation over $\mathcal{S}_{\text{3CNF-SAT}}$, a graph $\mathcal{G}_\varphi$ can be constructed such that φ can be satisfied if and only if $\mathcal{G}_\varphi$ is valid against $\mathcal{S}_{\text{3CNF-SAT}}$.

Then, consider the propositional formula φ in 3CNF form of n variables $x_1, \ldots, x_n$ and ℓ clauses. Then the formula is of the form $\varphi = C_1 \wedge C_2 \wedge \ldots C_\ell$, and each clause is of the form $C_i = l_{i1} \vee l_{i2} \vee l_{i3}$. We define the instance graph $\mathcal{G}_\varphi = \langle V_\varphi, E_\varphi \rangle$ as follows:

$$
\begin{aligned}
V_\varphi = {} & \{\texttt{:cla}, \texttt{:var}\} \cup \{C_i \mid i \in [1,\ell]\} \\
& \cup \{x_i \mid i \in [1,n]\} \cup \{\neg x_i \mid i \in [1,n]\} \\
& \cup \{\overline{x_i} \mid i \in [1,n]\} \\
E_\varphi = {} & \{(C_i, \texttt{:type}, \texttt{:cla}) \mid i \in [1,\ell]\} \\
& \cup \{(C_i, \texttt{:l}, l_{ij}) \mid i \in [1,n], j \in [1,3]\} \\
& \cup \{(\overline{x_i}, \texttt{:type}, \texttt{:var}) \mid i \in [1,n]\} \\
& \cup \{(\overline{x_i}, \texttt{:r}, x_i) \mid i \in [1,n]\} \cup \{(\overline{x_i}, \texttt{:r}, \neg x_i) \mid i \in [1,n]\} \\
& \cup \{(x_i, \texttt{:r}, x_i) \mid i \in [1,n]\} \cup \{(\neg x_i, \texttt{:r}, \neg x_i) \mid i \in [1,n]\}
\end{aligned}
$$

As a reference, Figure 4.4 shows the resulting graph $\mathcal{G}_\varphi$ if the formula is $\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$.

To show the reduction is correct, we will prove that φ can be satisfied if and only if $\mathcal{G}_\varphi$ is valid against $\mathcal{S}$. First, for the right direction, if φ can be satisfied there is a boolean assignment $\alpha : \text{var}(\varphi) \to \{0,1\}$ such that $\alpha(\varphi) = 1$.

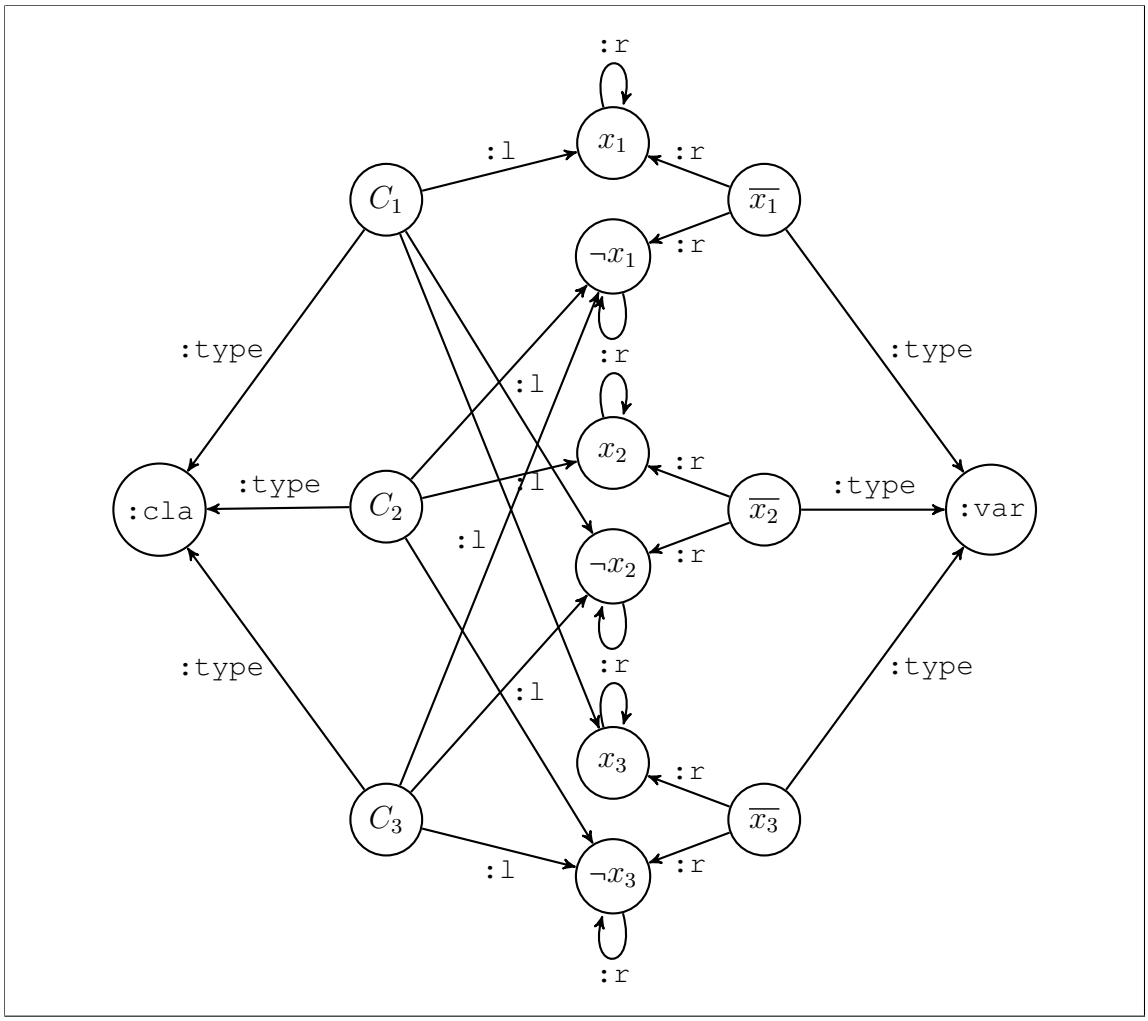

FIGURE 4.4. Instance graph $\mathcal{G}_\varphi$ reduction example.

Consider the shape assignment σ_α for $\mathcal{G}_\varphi$ and $\mathcal{S}$ defined as

$$\begin{aligned}\sigma_\alpha = &\{c(C_i) \mid i \in [1,\ell]\}\\ &\cup \{n(\overline{x_i}) \mid i \in [1,n]\}\\ &\cup \{\neg n(x_i) \mid \alpha(x_i) = 1\}\\ &\cup \{n(x_i) \mid \alpha(x_i) = 0\}\\ &\cup \{n(\neg x_i) \mid \alpha(x_i) = 1\}\\ &\cup \{\neg n(\neg x_i) \mid \alpha(x_i) = 0\}\end{aligned}$$

First, σ_α is valid since there is no pair of shape s and node v such that $\{s(v), \neg s(v)\} \subseteq \sigma_\alpha$ and it clearly contains the correct assignment for target queries over $\mathcal{G}_\varphi$, because all $c(C_i)$ and $n(\overline{x_i})$ are present. Second, the shape constraints must be satisfied. For every node $\overline{x_i}$, σ_α assigns the shape n to either one of its succesors x_i or $\neg x_i$, and $\neg n$ to the other. Since every pair of nodes x_i and $\neg x_i$ have self $:\mathtt{r}$ loops, either assigning n of its negation satisfies $\mathrm{def}(n) = \geq_1 :\mathtt{r}.\ n$. Therefore, $n(\overline{x_i}) \rightarrow [\mathrm{def}(n)]^{\overline{x_i},\mathcal{G}_\varphi,\sigma_\alpha} = 1$, and so does foe every $n(x_i)$, $n(\neg x_i)$, $\neg n(x_i)$ and $\neg n(\neg x_i)$.

Since φ is in CNF, then $\alpha(C_i) = 1$ for every clause C_i, meaning that for each one there is one literal (of three) where $\alpha(l_{i,j}) = 1$. If $l_{i,j}$ is positive, then its a variable x_k that is assigned to 1, and the corresponding node in $\mathcal{G}_\varphi$ is assigned as $\neg n(x_i)$. While if $l_{i,j}$ is negative, then its the negation of variable x_k that is assigned to 0, and the corresponding node in $\mathcal{G}_\varphi$ is assigned as $\neg n(\neg x_i)$. Therefore, every node C_i has at least one $:\mathtt{l}$ successor with $\neg n$ is assigned to it. Which means $c(C_i) \rightarrow [\mathrm{def}(c)]^{C_i,\mathcal{G}_\varphi,\sigma_\alpha} = 1$. Thus, proving that σ_α is faithful for $\mathcal{G}_\varphi$ and $\mathcal{S}$.

Now, for the left direction, if $\mathcal{G}_\varphi$ is valid against $\mathcal{S}$ then a faithful shape assignment σ for $\mathcal{G}_\varphi$ and $\mathcal{S}$ exists. Consider the boolean assignment α_σ defined as $\alpha_\sigma(x_i) = 1 \leftrightarrow \neg n(x_i) \in \sigma$.

For every clause C_i in φ, a corresponding node C_i in the $\mathcal{G}_\varphi$ exists with only three outgoing $:\mathtt{l}$ edges for other three nodes. Since σ is faithful, C_i is part of the target set for shape c, meaning $c(C_i) \rightarrow [\mathrm{def}(c)]^{C_i,\mathcal{G}_\varphi,\sigma_\alpha} = 1$, which implies thanks to $\mathrm{def}(c) = \geq_1 :\mathtt{l}.\ \neg n$ that one of these three $:\mathtt{l}$ successors is assigned $\neg n$. Let's call that node l, then $\neg n(l) \in \sigma$. Because of construction of $\mathcal{G}_\varphi$, l is a successor because l is a literal in C_i. If l is a positive literal then is also a variable, by definition $\neg n(l) \in \sigma$ and then $\alpha_\sigma(l) = 1$. If not, the positive literal $\neg l$ is also a node in $\mathcal{G}_\varphi$, where both l and $\neg l$ are successors to $\overline{x_k}$. $\overline{x_k}$ is in the target query for n, and therefore satisfies $\mathrm{def}(n)$ meaning l or $\neg l$ is assigned n. But $\neg n(l) \in \sigma$, then $n(\neg l) \in \sigma$. And since σ is a valid shape assignment, $\neg n(\neg l) \notin \sigma$, then $\alpha_\sigma(\neg l) = 0$ and $\alpha_\sigma(l) = 1$. Either case, for every clause C_i in φ a literal l exists such that $\alpha_\sigma(l) = 1$, and $\alpha_\sigma(C_i) = 1$. Thus $\alpha_\sigma(\varphi) = 1$, which means φ can be satisfied.

This finally proves that 3CNF-SAT can be reduced to validation of $\mathcal{S}_{\text{3CNF-SAT}}$, where $G_{\mathcal{S}_{\text{3CNF-SAT}}}$ does not contain an negative cycle.

□

This shows that even recursive schemas with very simple dependency graphs and without negative cycles are as hard as 3CNF-SAT. But more specifically, it shows that the interplay between targeted shapes and negation is also key in for tractability in the validation problem. Although, the referenced definitions for stratified and strictly stratified schemas already considered this aspect, both of them are specified over schemas with single shape with target query as generalization of multiple shapes with target queries. Any schema can be transformed to fit this form, by simply adding a new shape and redefining target queries accordingly, as the right schema in Figure 4.3 that introduces the shape s_0 as the single targeted shape. This consideration translates into adding edges into $G_{\mathcal{S}}$ that represent target query dependencies to be considered in the whole schema relation. Thus, following the stratified family definition, the equivalent schema for the one used in the reduction is indeed stratified but not strictly stratified, which explains intractability.

Thus far, three tractable fragments have been introduced, and each one avoids intractability in different and almost exclusive ways. $\mathcal{L}^{\text{non-rec}}$ avoids recursion all together and is the easiest case. While $\mathcal{L}^{+}_{\vee}$ avoids the use of negation which simplifies how to handle cyclic dependencies. Finally, $\mathcal{L}^{s}$ takes into account the interplay of negation, recursion and targets. What it is common between the three, is that neither handle negations in the scope of a cycle in any way, nor define a well behaved negative cycle which will later show actually exists.

As for SPARQL expressivity, the results are negative. Since validation of full SHACL in data complexity is NP-hard, full SHACL cannot be reduced to SPARQL evaluation (which is PTIME in data complexity, as shown in (Pérez, Arenas, & Gutiérrez, 2009)). Even if we focus on the introduced fragments for which it is known validation is tractable, we have a negative result. Schemas in $\mathcal{L}^{s}$ and $\mathcal{L}^{+}_{\vee}$ exists such that they surpass SPARQL expressivity, as shown in (Corman et al., 2019) (Section 4). Then if the schema is recursive, it is not

possible in general to retrieve target nodes violating a given shape by issuing a single SPARQL query. This means that some extra computation (in addition to SPARQL query evaluation) needs to be performed, in memory.

Even tough the general case has a negative result, we insist on the use of SPARQL query evaluation for validation as most RDF datasets are only accessible through SPARQL endpoints, and that it may be reasonable to consider using multiple queries instead of one. The latter may bring a trade-off that simplifies the complexity of query evaluation. This is exactly the case for an all general recursive schema validation approach that uses multiple queries whose results then are processed in memory. We introduce a revision of this approach in the next section, as we alter it into a tractable algorithm in Section 6.

5. RULE PATTERNS

This section provides a revision of the proposed approach in (Corman et al., 2019) to validate arbitrary SHACL schemas over a SPARQL endpoint. Looking for a satisfying assignment using brute-force only is definitely out of the question. Instead, the problem can be attacked by reducing validation to checking if a propositional formula can be satisfied, possibly leveraging the optimization techniques of a SAT solver.

As previously shown, when testing if a shape s can be assigned to a node v, the negative answer could be tested locally and easily, but it may also depend on the assignment of another shape over a neighbor node v'. How these dependencies chain and link between them defines the complexity of the whole validation problem. Then, the rule pattern approach arises as a way to keep track of all dependencies between potential shape assignments for nodes, both acyclical and cyclical. And does so by constructing propositional formulas that encode these dependencies, and considering the whole set of formulas as a satisfaction problem. If the set of formulas can be satisfied, then there is a way to assign shapes following the dependencies, and vice versa.

Specifically, given a graph $\mathcal{G}$ to validate against a shape schema $\mathcal{S}$, the road map of rule pattern approach is as follows. First, a *normal form* for shape schemas is defined and it is expected that the target schema follows this form. Second, one SPARQL query is associated to each shape in the normalized schema, such that it will match all nodes in $\mathcal{G}$ that can potentially be assigned to the corresponding shape. Third, from the evaluation of these queries over the graph database we construct a set of rules of the form $l_0 \wedge .. \wedge l_n \rightarrow s(v)$, where each l_i is either $s_i(v_i)$ or $\neg s_i(v_i)$, for some $s_i \in S$ and $v_i \in V_{\mathcal{G}}$. Intuitively, a rule such as $s_1(v_1) \wedge \neg s_2(v_2) \rightarrow s(v)$ means that, if node v_1 conforms to shape s_1 and node v_2 does not conform to shape s_2, then node v conforms to shape s. These rules alone are not sufficient for a sound validation algorithm, so we complement them with additional rules encoding targeted nodes, and other necessary information. Finally, by testing if the whole set of propositional formulas generated can be satisfied, the verification of $\mathcal{G}$ over $\mathcal{S}$ can be answered. Correctness of this method is proven in (Corman et al., 2019).

This approach can handle validations with respect to either total or partial assignments. For validation with respect to partial assignments the set of rules can be satisfied under 3-valued (Kleene's) logic, while for validation with respect to total assignments, the set of rules can be satisfied under standard (2-valued) propositional logic. We will focus on the latter alternative.

As shown in (Corman et al., 2019), the machinery of this approach could be used to design more efficient algorithms for certain tractable fragments, without the need of a SAT solver. By reviewing this approach and analyzing the form and nature of the constructed formulas, we have found a new bigger and tractable fragment of SHACL schemas. To introduce this fragment on Section 6, we spend the rest of this section to review the necessary notation, definitions for each of its components and changes made in this version.

Normal form. A shape schema $\mathcal{S} = \langle S, \mathrm{targ}, \mathrm{def} \rangle$ is *in normal form* if the set S of shape names can be partitioned into three sets S^+, S^- and S^{NEQ}, such that for each $s \in S^+$ (resp. $s \in S^-$ and $s \in S^{\text{NEQ}}$), $\mathrm{def}(s)$ verifies ϕ_{s^+} (resp. ϕ_{s^-} and $\phi_{s^{\text{NEQ}}}$) in the following grammar:

$$
\begin{aligned}
\phi_{s^+} &::= \alpha \mid \geq_n r.\alpha \mid \phi_{s^+} \wedge \phi_{s^+} \\
\alpha &::= \top \mid I \mid s \\
\phi_{s^-} &::= \neg s \\
\phi_{s^{\text{NEQ}}} &::= \neg \mathrm{EQ}(r_1, r_2)
\end{aligned}
$$

It is easy to verify that a shape schema can be transformed in linear time into an equivalent normalized one, by introducing fresh shape names (without target). We mean equivalent by that both schemas validate exactly the same graphs, with exactly the same target violations. This is a revised version of the normal form proposed in (Corman et al., 2019), where the main change was the normalized use of negation. Now negation can only be used through a single shape definition, leaving all other constraints as "positive".

SPARQL queries. Normalization allows us to associate a SPARQL query $q_{\mathrm{def}(s)}$ to each shape name in the normalized schema. The purpose is that the query $q_{\mathrm{def}(s)}$ retrieves all nodes that *potentially* validates $\mathrm{def}(s)$, and also the corresponding neighboring nodes to which satisfaction may depend on.

For instance, let $\mathrm{def}(s_0) = (\geq_1 \texttt{:p1}.s_1) \wedge (\geq_1 \texttt{:p2}.s_2)$. For a node to be assigned s_0, it needs to have at least one `:p1`-neighbor and one one `:p2`-neighbor. Then, the following query filters out any nodes that do not have these local properties:[1]

$$q_{\mathrm{def}(s_0)} = \texttt{SELECT ?x ?y1 ?y2 WHERE } \{\texttt{?x :p1 ?y1 . ?x :p2 ?y2 }\}$$

The inductive definition of $q_{\mathrm{def}(s)}$ can be found in (Corman et al., 2019) (Figure 3), we omit it as it does not change for our purposes and can also be applied over the updated normal form. A notable aspect of this translation is that it originates altogether such simpler queries to be processed by the SPARQL endpoint than the one presented for non-recursive schemas.

Rule patterns. The next step consists in generating a set of propositional formulas, based on the evaluation of the queries that have just been defined. To generate such formulas, we associate a *rule pattern* $p_{\mathrm{def}(s)}$ to each shape $s \in S$. This rule pattern is of the form $l_1 \wedge .. \wedge l_n \rightarrow s(\texttt{?x})$, where each l_i is either $\top$, $s_i(w_i)$ or $\neg s_i(w_i)$, for some shape $s_i \in S$ and variable w. The definition of $p_{\mathrm{def}(s)}$ is inductive on the structure of $\mathrm{def}(s)$, and can be found in (Corman et al., 2019) (Figure 4).

Continuing the example above, if $\mathrm{def}(s_0) = (\geq_1 \texttt{:p1}.s_1) \wedge (\geq_1 \texttt{:p2}.s_2)$, then:

$$q_{\mathrm{def}(s_0)} = \texttt{SELECT ?x ?y1 ?y2 WHERE } \{\texttt{?x :p1 ?y1 . ?x :p2 ?y2 }\}$$

$$p_{\mathrm{def}(s_0)} = s_1(\texttt{?y1}) \wedge s_2(\texttt{?y2}) \rightarrow s_0(\texttt{?x})$$

Rule formulas. Each rule pattern $p_{\mathrm{def}(s)}$ is then instantiated as propositional formulas using the answers of $q_{\mathrm{def}(s)}$ over the SPARQL endpoint, which yields a set $[\![p_{\mathrm{def}(s)}]\!]^{\mathcal{G}}$ of

[1] We omit a trivial `FILTER (?y1 = ?y1 AND ?y2 = ?y2)` for readability.

formulas, called *rule formulas*. Intuitively, each rule represents a shape assignment dependency from the neighbors of a node v, when trying to validate that node against a shape s. This step transformes potential dependencies in the schema, into the actual assignment dependencies that appear in a graph. For instance, assume that the endpoint returns the following mappings for $q_{\mathrm{def}(s_0)}$:

$$\begin{aligned}
[\![q_{\mathrm{def}(s_0)}]\!]^{\mathcal{G}} = \{\{\texttt{?x} &\mapsto v_0, \texttt{?y1} \mapsto v_1, \texttt{?y2} \mapsto v_2\},\\
&\{\texttt{?x} \mapsto v_0, \texttt{?y1} \mapsto v_3, \texttt{?y2} \mapsto v_4\}\}\\
[\![p_{\mathrm{def}(s_0)}]\!]^{\mathcal{G}} = \{&s_1(v_1) \wedge s_2(v_2) \rightarrow s_0(v_0),\\
&s_1(v_3) \wedge s_2(v_4) \rightarrow s_0(v_0)\}
\end{aligned}$$

The first rule means if v_1 verifies $\mathrm{def}(s_1)$, and v_2 verifies $\mathrm{def}(s_2)$, then one can infer that v_0 verifies $\mathrm{def}(s_0)$ (and similarly for the second rule). For every rule formula $l_1 \wedge .. \wedge l_n \rightarrow l$, we will call l the *head* of the formula, while $l_1 \wedge .. \wedge l_n$ is the *body* of the formula. Also, we use $[\![p_{\mathcal{S}}]\!]^{\mathcal{G}}$ to designate the set of all generated rule formulas, i.e. $[\![p_{\mathcal{S}}]\!]^{\mathcal{G}} = \bigcup_{s \in S} [\![p_{\mathrm{def}(s)}]\!]^{\mathcal{G}}$.

So far, rule formulas capture the shape constrains definitions grounded to possible nodes. Unfortunately, these are not enough to reflect the whole shape assignment problem. So additional formulas are introduced to achieve that. These are *complement formulas*, *target formulas* and *non-retrieval formulas*.

Complement formulas. So far, with a rule $s_1(v_1) \wedge s_2(v_2) \rightarrow s_0(v_0)$, we are capturing the idea that v_0 must be assigned shape s_0 whenever v_1 is assigned s_1 and v_2 is assigned s_2. But we also need to encode that the only way for v_0 to be assigned shape s_0 is to satisfy one of these rules. If there is just one rule with $s_0(v_0)$ as its head, we only need to extend our set of rules with $s_0(v_0) \rightarrow s_1(v_1) \wedge s_2(v_2)$. But for more generality, we construct a second set $[\![p^{\leftarrow}_{\mathcal{S}}]\!]^{\mathcal{G}}$ of propositional formulas called *complement formulas* as follows. For every literal $s(v)$ that appears as the head of a rule $C \rightarrow s(v)$ in $[\![p_{\mathcal{S}}]\!]^{\mathcal{G}}$, let $C_1 \rightarrow s(v), .., C_\ell \rightarrow s(v)$ be all the rules that have $s(v)$ as head. Then we extend $[\![p^{\leftarrow}_{\mathcal{S}}]\!]^{\mathcal{G}}$ with the formula $s(v) \rightarrow (C_1 \vee .. \vee C_\ell)$. Now, for every complement formula

$l \to (C_1 \vee .. \vee C_\ell)$, we will call l the *head* of the formula, while $C_1 \vee .. \vee C_\ell$ is the *body* of the formula.

Target formulas. Next, we add the information about all target nodes, with the set $[\![t_\mathcal{S}]\!]^\mathcal{G}$ of (atomic) formulas, defined by $[\![t_\mathcal{S}]\!]^\mathcal{G} = \{s(v) \mid s \in S, v \in [\![\mathrm{targ}(s)]\!]^\mathcal{G}\}$.

Non-retrieval formulas. Finally, we use a last set of formulas to ensure that the algorithm is sound and complete. Intuitively, the query $q_{\mathrm{def}(s)}$ retrieves all nodes that may verify shape s (bound to variable `?x`). But evaluating $q_{\mathrm{def}(s)}$ also provides information about the nodes that are *not* retrieved: namely that they *cannot* verify shape s. A first naive idea is to extend our set of propositional formulas with every literal $\neg s(v)$ for which $[\![q_{\mathrm{def}(s)}]\!]^\mathcal{G}$ does not contain any mapping where v is bound to `?x`. But this may require retrieving all nodes in $\mathcal{G}$ beforehand, which is inefficient. One can do better, by considering only combinations of shapes and nodes that are already in our rules. We thus construct another set $[\![a_\mathcal{S}]\!]^\mathcal{G}$ of facts called *non-retrieval formulas*. It contains all literals of the form $\neg s(v)$ such that: $\neg s(v)$ or $s(v)$ appears in some formula in $[\![p_\mathcal{S}]\!]^\mathcal{G} \cup [\![t_\mathcal{S}]\!]^\mathcal{G}$, and $s(v)$ is not the head of any formula in $[\![p_\mathcal{S}]\!]^\mathcal{G}$ (i.e. there is no rule of the form $\psi \to s(v)$ in $[\![p_\mathcal{S}]\!]^\mathcal{G}$).

Validation formula set. Let $\Gamma_{\mathcal{G},\mathcal{S}}$ be the *validation formula set*, which is the union of all the sets of formulas constructed so far: $\Gamma_{\mathcal{G},\mathcal{S}} = [\![p_\mathcal{S}]\!]^\mathcal{G} \cup [\![p_\mathcal{S}^{\leftarrow}]\!]^\mathcal{G} \cup [\![t_\mathcal{S}]\!]^\mathcal{G} \cup [\![a_\mathcal{S}]\!]^\mathcal{G}$. We treat $\Gamma_{\mathcal{G},\mathcal{S}}$ as a set of propositional formulas over the set $\{s(v) \mid s \in S, v \in V_\mathcal{G}\}$ of propositions. As shown in (Corman et al., 2019), the set $\Gamma_{\mathcal{G},\mathcal{S}}$ is polynomial in the size of the evaluation of all queries $\mathrm{def}(s)$ and $\mathrm{targ}(s)$. And more importantly, $\Gamma_{\mathcal{G},\mathcal{S}}$ can be satisfied if and only if $\mathcal{G}$ is valid against $\mathcal{S}$.

Hence, validity of schemas over graphs can be checked by constructing $\Gamma_{\mathcal{G},\mathcal{S}}$ and checking if it can be satisfied with a SAT solver. This algorithm matches the NP upper bound in data complexity mentioned earlier, since each of $[\![q_{\mathrm{def}(s)}]\!]^\mathcal{G}$ and $[\![\mathrm{targ}(s)]\!]^\mathcal{G}$ can be computed in polynomial time, when $\mathcal{S}$ is considered to be fixed, and thus the set $\Gamma_{\mathcal{G},\mathcal{S}}$ of rules can be computed in polynomial time in data complexity.

Rule pattern evaluation reflects the dependencies between shape assignments that will form when checking satisfaction of shape constraints. As an example, we show in

Table 5.1 the produced validation formula set $\Gamma_{\mathcal{G},\mathcal{S}}$ for the schema and example graph showed in the proof for Proposition 4.1. Cyclical dependencies appear as for every pair $n(x_i)$ and $n(\neg x_i)$, while other assignments (like $c(C_j)$ and $n(\overline{x_i})$) depend on either those assignments or their negation. The original approach proposed would then make use of an external SAT solver to check satisfiability for this set.

Contrarily, it was also shown in (Corman et al., 2019) that for some tractable fragments, as $\mathcal{L}^{\text{non-rec}}$, $\mathcal{L}^{+}_{\vee}$ and $\mathcal{L}^{s}$, an in-memory processing of these formulas by applying inference can achieve tractability. We continued developing this idea, by studying the resulting set of formulas that rule patterns generates and if by applying logical resolution rules a sound algorithm can be reached.

For example, literals in $[\![t_{\mathcal{S}}]\!]^{\mathcal{G}}$ and $[\![a_{\mathcal{S}}]\!]^{\mathcal{G}}$ that need to be true give information about formulas that may already be satisfied in $[\![p_{\mathcal{S}}]\!]^{\mathcal{G}}$ and $[\![p^{\leftarrow}_{\mathcal{S}}]\!]^{\mathcal{G}}$, and therefore do not contribute relevant information, or others formulas that may simplify and result in new information about assignments. Table 5.2 shows an example of newly derived formulas by taking these literals as true.

Unfortunately, and as one might expect, in the general case this simplification will need help of a SAT solver anyways. But, if we restrict the properties for a certain recursive fragment, we find that this process arrives to a sound and complete validation process. This is shown in Section 6.

$[\![p_{\mathcal{S}}]\!]^{\mathcal{G}}$		$[\![p_{\mathcal{S}}^{\leftarrow}]\!]^{\mathcal{G}}$
$n(x_1) \to n(x_1)$	$\neg n(x_1) \to c(C_1)$	$n(x_1) \to n(x_1)$
$n(\neg x_1) \to n(\neg x_1)$	$\neg n(\neg x_2) \to c(C_1)$	$n(\neg x_1) \to n(\neg x_1)$
$n(x_2) \to n(x_2)$	$\neg n(x_3) \to c(C_1)$	$n(x_2) \to n(x_2)$
$n(\neg x_2) \to n(\neg x_2)$	$\neg n(\neg x_1) \to c(C_2)$	$n(\neg x_2) \to n(\neg x_2)$
$n(x_3) \to n(x_3)$	$\neg n(x_2) \to c(C_2)$	$n(x_3) \to n(x_3)$
$n(\neg x_3) \to n(\neg x_3)$	$\neg n(\neg x_3) \to c(C_2)$	$n(\neg x_3) \to n(\neg x_3)$
$n(x_1) \to n(\overline{x_1})$	$\neg n(\neg x_3) \to c(C_2)$	$n(\overline{x_1}) \to n(x_1) \vee n(\neg x_1)$
$n(\neg x_1) \to n(\overline{x_1})$	$\neg n(\neg x_2) \to c(C_3)$	$n(\overline{x_2}) \to n(x_2) \vee n(\neg x_2)$
$n(x_2) \to n(\overline{x_2})$	$\neg n(\neg x_3) \to c(C_3)$	$n(\overline{x_3}) \to n(x_3) \vee n(\neg x_3)$
$n(\neg x_2) \to n(\overline{x_2})$		$c(C_1) \to \neg n(x_1) \vee \neg n(\neg x_2) \vee \neg n(x_3)$
$n(x_3) \to n(\overline{x_3})$		$c(C_2) \to \neg n(\neg x_1) \vee \neg n(x_2) \vee \neg n(\neg x_3)$
$n(\neg x_3) \to n(\overline{x_3})$		$c(C_3) \to \neg n(\neg x_1) \vee \neg n(\neg x_2) \vee \neg n(\neg x_3)$

$[\![t_{\mathcal{S}}]\!]^{\mathcal{G}}$		$[\![a_{\mathcal{S}}]\!]^{\mathcal{G}}$
$n(\overline{x_1})$	$c(C_1)$	$\varnothing$
$n(\overline{x_2})$	$c(C_1)$	
$n(\overline{x_3})$	$c(C_3)$	

TABLE 5.1. Example of constructed $\Gamma_{\mathcal{G},\mathcal{S}}$ for reduction in Figure 4.4

$[\![p_{\mathcal{S}}]\!]^{\mathcal{G}}$	$[\![p_{\mathcal{S}}^{\leftarrow}]\!]^{\mathcal{G}}$		$[\![t_{\mathcal{S}}]\!]^{\mathcal{G}}$
$n(x_1) \to n(x_1)$	$n(x_1) \to n(x_1)$	$n(x_1) \vee n(\neg x_1)$	$n(\overline{x_1})$
$n(\neg x_1) \to n(\neg x_1)$	$n(\neg x_1) \to n(\neg x_1)$	$n(x_2) \vee n(\neg x_2)$	$n(\overline{x_2})$
$n(x_2) \to n(x_2)$	$n(x_2) \to n(x_2)$	$n(x_3) \vee n(\neg x_3)$	$n(\overline{x_3})$
$n(\neg x_2) \to n(\neg x_2)$	$n(\neg x_2) \to n(\neg x_2)$	$\neg n(x_1) \vee \neg n(\neg x_2) \vee \neg n(x_3)$	$c(C_1)$
$n(x_3) \to n(x_3)$	$n(x_3) \to n(x_3)$	$\neg n(\neg x_1) \vee \neg n(x_2) \vee \neg n(\neg x_3)$	$c(C_2)$
$n(\neg x_3) \to n(\neg x_3)$	$n(\neg x_3) \to n(\neg x_3)$	$\neg n(\neg x_1) \vee \neg n(\neg x_2) \vee \neg n(\neg x_3)$	$c(C_3)$

TABLE 5.2. Filtered and simplified formulas from Figure 5.1

6. A TRACTABLE RECURSIVE FRAGMENT

In this section, we will define a SHACL fragment that is an extension of previous identified tractable fragments, and then we will proceed to show it is tractable as well by stating an algorithm using the rule pattern approach. First, we will define the needed terms to identify it, and then we will propose the tractable algorithm.

6.1. Definition and expressiveness

As for the rest of this document, for all definitions we consider $\mathcal{S} = \langle S, \mathrm{targ}, \mathrm{def} \rangle$ as a shape schema, $G_{\mathcal{S}}$ its corresponding dependency graph and $\mathcal{G}$ a graph database to validate.

First, a shape $s \in S$ is considered *terminal* if and only if one of the following holds: s is a sink node in $G_{\mathcal{S}}$, i.e. s has no outgoing edges; or s is not part in of a cycle in $G_{\mathcal{S}}$, and all of its node successors are also terminal shapes. Terminal shapes represent all shapes that can be actually completely validated without cyclical dependencies, as for the case of $\mathcal{L}^{\text{non-rec}}$. Furthermore, if $\mathcal{S} \in \mathcal{L}^{\text{non-rec}}$, then all shapes in S are terminal.

This introduces the idea of detecting a two set partition in the shape schema: terminal shapes, and non-terminal shapes. Our proposal considers that depending on how the non-terminal shapes are specified and relate between each other, then tractability can be assured.

Specifically, we define consistent SHACL ($\mathcal{L}^{\text{cons}}$) as the fragment of shape schemas that are *consistent*. A shape schema $\mathcal{S}$ is ***consistent*** if and only if it is in normal form (as defined in Section 5) and a labeling function $\tau : S \to \{0, 1/2, 1\}$ exists such that:

- For any shape $s \in S$, $\tau(s) = 1/2$ if and only if s is a terminal shape.
- For any shape $s \in S$, if $\mathrm{targ}(s) \neq \bot$ then $\tau(s) \geq 1/2$.
- For every pair of non-terminal shapes $s, s' \in S$:
  - if there is a negative edge between s and s' in $G_{\mathcal{S}}$ then $\tau(s) = 1 - \tau(s')$;
  - if there is a positive edge between then, then $\tau(s) = \tau(s')$

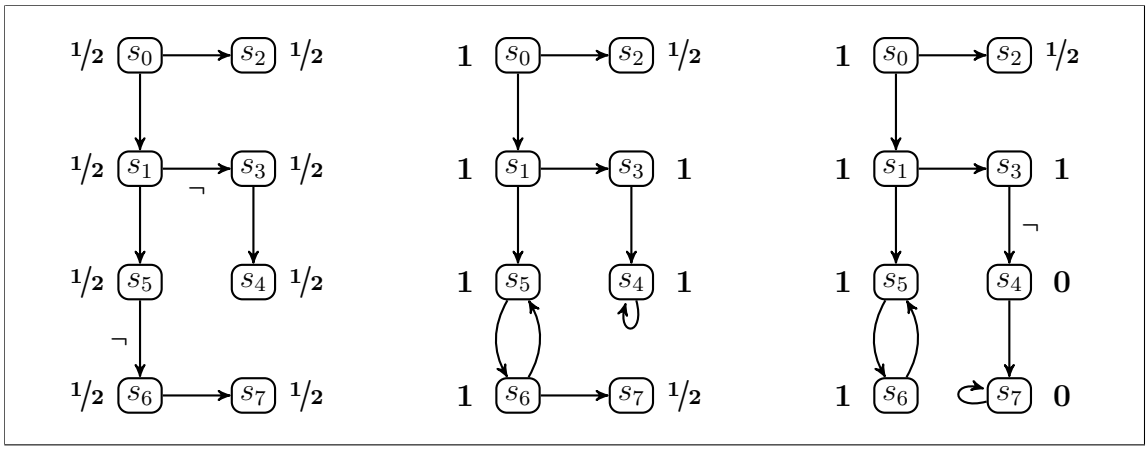

FIGURE 6.1. Consistent labeling for acyclic, positive and strictly stratified schemas.

This definition extends the idea well behaved dependencies found in the other tractable fragments. On one hand, identifies terminal shapes as the ones where satisfaction checking is direct (label $\mathbf{1/2}$), and thus identifies the recursive portions of the schema (labels $\mathbf{1}$ and $\mathbf{0}$). And on the other, establishes conditions that need to be met considering negation in dependencies and target queries dependencies. Figure 6.1 shows some labeling examples for dependency graphs of tractable schemas: $\mathcal{L}^{\text{non-rec}}$ at the left, $\mathcal{L}^{+}_{\vee}$ in the middle, and $\mathcal{L}^{s}$ at the right. Proposition 6.1 shows that consistent SHACL indeed extends the tractable fragments revised in Section 4 and also captures strictly more schemas.

Proposition 6.1. $(\mathcal{L}^{\textit{non-rec}} \cup \mathcal{L}^{+}_{\vee} \cup \mathcal{L}^{s}) \subsetneq \mathcal{L}^{\textit{cons}}$

PROOF. We will show for each fragment separately that all contained schemas are also consistent. Then, by example we can prove a consistent schema exists that is in neither of the other tractable mentioned fragments.

Consider $\mathcal{S} = \langle S, \text{targ}, \text{def} \rangle \in \mathcal{L}^{\text{non-rec}}$. By definition, all $s \in S$ are terminal shapes in $G_{\mathcal{S}}$. Otherwise, a cycle should exists in $G_{\mathcal{S}}$. Therefore, an equivalent schema S' can be constructed introducing new shapes such that S' is in normal form, and the constant labeling function $\tau(s) = 1/2$ assigns all shapes the same value as they are all terminal. τ suffices the needed properties trivially, thus $\mathcal{S} \in \mathcal{L}^{\text{cons}}$.

Now, consider $\mathcal{S} = \langle S, \text{targ}, \text{def} \rangle \in \mathcal{L}^+_\vee$. Since this fragment uses disjunction freely, it can not be directly compared. But, to match semantics, every use of disjunction, say as $\text{def}(s) = s_1 \vee s_2$, can be converted to $\text{def}(s) = \neg((\neg s_1) \wedge (\neg s_2))$. This introduces negation to the schema only in the scope of disjunctions, but it may appear in the path of a cycle. Because the dependency between s and potential shapes s_1 and s_2 entails two negations, the consistency labeling of the schema can be achieved easily. An equivalent schema $\mathcal{S}'$ can be constructed so that it is in normal form and that takes into account the normalization of converted disjunction, that may be as follows for the previous example: $\text{def}(s) = \neg s'$, $\text{def}(s') = s'_1 \wedge s'_2$, $\text{def}(s'_1) = \neg s_1$ and $\text{def}(s'_2) = \neg s_2$. By doing so, a function τ can be defined such that for every terminal shape original to $\mathcal{S}$ are labeled as $1/2$, and all non-terminal shapes original to $\mathcal{S}$ are labeled as 1, while introduced shapes in $\mathcal{S}'$ for the normalization of disjunctions are labeled as: $\tau(s') = 0$, $\tau(s'_1) = 0$ and $\tau(s'_2) = 0$ (shape names are in reference to the used example). τ makes $\mathcal{S}'$ consistent since: there is no use of negation apart from disjunctions, and therefore labeling 1 or $1/2$ to shapes original to $\mathcal{S}$ is fitting; and since the only labeled shapes as 0 are added shapes enclosed by negative edges, they do not violate the target query condition and also have fitting labeling with their neighbors in $G_\mathcal{S}$. Thus, $\mathcal{S} \in \mathcal{L}^+_\vee$.

Now, consider $\mathcal{S} = \langle S, \text{targ}, \text{def} \rangle \in \mathcal{L}^s$. The definiton in (Corman et al., 2018c) takes as assumption that schemas have only one target query defined for a single shape s_0 and node v_0. With that in mind, and taking an equivalent schema $\mathcal{S}'$ in normal form, we can define a consistent labeling function τ as follows. First, $\tau(s) = 1/2$ for all terminal shapes in the schema, if s_0 is not terminal, then $\tau(s_0) = 1$. Then, for every non-terminal shape s such that a path from s_0 to s exists, if the path is negative and then number of negations in the path is odd then $\tau(s) = 0$, for all other cases with the existence of a path, $\tau(s) = 1$. Since there is no guarantee that all shapes are contained in the same component as s_0 in $G_{\mathcal{S}'}$, the same labeling definition can be extended by taking an arbitrary node s'_0 in a separated component as fixed such that $\tau(s'_0) = 1$ and applying the previous definition with that shape as reference. Labeling function τ directly satisfies the target query and terminal shapes labeling. For every pair of non-terminal shapes s' and s'' that are connected through

an edge in $G_{\mathcal{S}'}$, this edge may be positive or negative. In the positive case and if both are contained in the same component as s_0, because $\mathcal{S}' \in \mathcal{L}^s$, then from s_0 both s' and s'' must be connected through positive paths, or by negative paths with the same amount of negations in it. Therefore, in either case $\tau(s') = \tau(s'')$. Now, if they are connected by a negative edge and both are contained in the same component as s_0, all paths from s_0 and s'' must pass by s'. Then, independent if the paths between s_0 and s' have negation of not, then a path between s_0 and s'' has an extra negation which τ would label s'' the opposite as s'. The previous arguments can be extended if s' and s'' are not contained in the same component as s_0, but have s'_0 as a reference. Thus, τ is a valid consistent labeling.

Now, we define the counterexample. Consider the schema $\mathcal{S}^{\text{cons}} = \langle S, \text{targ}, \text{def} \rangle$, defined as follows[1]:

$$
\begin{aligned}
& S = \{s_1, s_2, s_3, s_4\} \\
& \text{targ}(s_1) = v_1 \qquad && \text{def}(s_1) = \geq_1 p_1.s_2 \\
& \text{targ}(s_2) = v_2 \qquad && \text{def}(s_2) = \neg s_3 \\
& \text{targ}(s_3) = \bot \qquad && \text{def}(s_3) = \geq_1 p_2.s_4 \\
& \text{targ}(s_4) = \bot \qquad && \text{def}(s_4) = \neg s_1
\end{aligned}
$$

$\mathcal{S}^{\text{cons}}$ is consistent since a consistent labeling function τ for $\mathcal{S}$ can be defined. Let $\tau(s) = 1$ for $s \in \{s_1, s_2\}$ and $\tau(s) = 0$ for $s \in \{s_3, s_4\}$. Since all shapes are part of the s_1, s_2, s_3, s_4 cycle, there are no terminal shapes to be labeled $1/2$. s_1 and s_2 are correctly labeled since both have target queries. τ also takes into account the positive edge between s_1 and s_2; the positive edge between s_3 and s_4; the negative edge between s_2 and s_3; and the negative edge between s_4 and s_1. Its clear $\mathcal{S} \notin \mathcal{L}^{\text{non-rec}} \cup \mathcal{L}^+_\vee \cup \mathcal{L}^s$ since there is negation used in a dependency cycle in $G_{\mathcal{S}}$. Thus, $\mathcal{S}^{\text{cons}} \in \mathcal{L}^{\text{cons}}$ but $\mathcal{S}^{\text{cons}} \notin \mathcal{L}^{\text{non-rec}} \cup \mathcal{L}^+_\vee \cup \mathcal{L}^s$.

□

The last part of the proof for the previous proposition showed that consistent SHACL does include dependencies cycles with negations in them. This, directly means that not

[1] The target definition for shapes s_1 and s_2 is an abuse of notation to indicate that their evaluation will result in specific nodes v_1 and v_2.

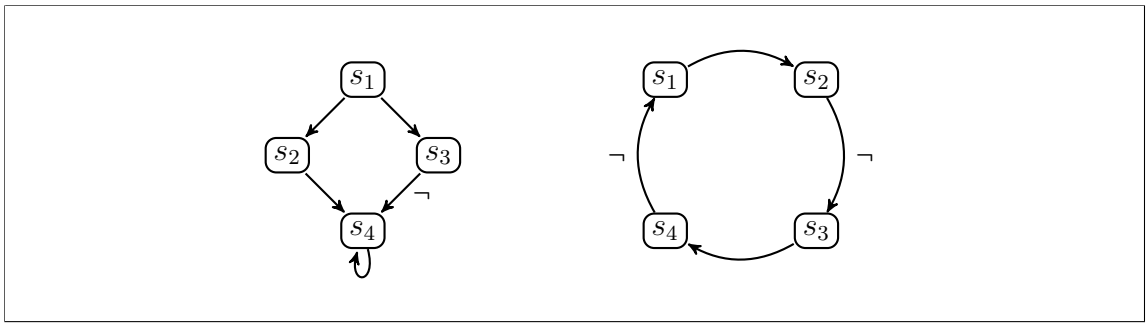

FIGURE 6.2. Counterexamples between stratified and consistent schemas.

all consistent schemas are necessarily stratified, but it also turns out that not all stratified schemas are consistent. Proposition 6.2 shows this, meaning that consistent schemas form a separate family of schemas.

Proposition 6.2. *$\mathcal{L}^{strat}$ and $\mathcal{L}^{cons}$ are incomparable. i.e. $\mathcal{L}^{strat} \notin \mathcal{L}^{cons}$ and $\mathcal{L}^{cons} \notin \mathcal{L}^{strat}$.*

PROOF. To show that $\mathcal{L}^{\text{cons}} \notin \mathcal{L}^{\text{strat}}$, the same counterexample $\mathcal{S}^{\text{cons}}$ used in the proof for Proposition 6.1 can be used. The right graph in Figure 6.2 shows the dependency graph of this counterexample. Because the graph is a negative cycle, $\mathcal{S}^{\text{cons}}$ cannot be stratified since a valid stratum labeling cannot be defined.

To show that $\mathcal{L}^{\text{strat}} \notin \mathcal{L}^{\text{cons}}$, we define $\mathcal{S}^{\text{strat}} = \langle S, \text{targ}, \text{def} \rangle$, as follows[2]:

$$
\begin{aligned}
&S = \{s_1, s_2, s_3, s_4\} \\
&\text{targ}(s_1) = v_1 \qquad && \text{def}(s_1) = \ (\geq_1 p_1.s_2) \wedge (\geq_1 p_1.s_3) \\
&\text{targ}(s_2) = \bot && \text{def}(s_2) = \geq_1 p_2.s_4 \\
&\text{targ}(s_3) = \bot && \text{def}(s_3) = \geq_1 p_2.\neg s_4 \\
&\text{targ}(s_4) = \bot && \text{def}(s_4) = \geq_1 p_3 s_4
\end{aligned}
$$

The left dependency graph in Figure 6.2 corresponds to $G_{\mathcal{S}^{\text{strat}}}$. $\mathcal{S}^{\text{strat}} \in \mathcal{L}^{\text{strat}}$ cause the number labeling $\text{str} : S \to \mathbb{N}$ can be defined as follows: $\text{str}(s_1) = 2$, $\text{str}(s_2) = 1$, $\text{str}(s_3) = 1$, $\text{str}(s_4) = 0$. This labeling successfully implies a well defined stratum in S, thus $\mathcal{S}^{\text{strat}}$ is stratified. Now, let's assume a consistent labeling function τ does exists for $\mathcal{S}^{\text{strat}}$. First,

[2] Again, the target definition for shape s_1 is an abuse of notation to indicate that its evaluation will result in the specific nodes v_1.

since there are no terminal shapes in $G_{\mathcal{S}^{\text{strat}}}$, then $\tau(s) \neq 1/2$ for every shape. Second, $\text{targ}(s_1) \neq \perp$ then $\tau(s_1) = 1$. Third, positive edges exists such that $1 = \tau(s_1) = \tau(s_2) = \tau(s_3) = \tau(s_4)$. But, a negative edge exists between s_3 and s_4, but $\tau(s_3) \neq 1 - \tau(s_4)$. Then a contradiction is met and τ cannot exists. Thus, $\mathcal{S}^{\text{strat}} \notin \mathcal{L}^{\text{cons}}$.

□

The fact that consistent and stratified schemas form non-inclusive families of schemas raises the question if the validation problem by partial and total assignments coincide for consistent schemas. After all, the main case where they did not coincide was because of the presence of negation in a cycle, which stratified schemas avoid. Proposition 6.3 answers this question with a positive.

Proposition 6.3. *Let $\mathcal{S}$ be a consistent shape schema and $\mathcal{G}$ a graph. Then there exists a partial faithful assignment for $\mathcal{G}$ and $\mathcal{S}$ if and only if there exists a total faithful assignment for $\mathcal{G}$ and $\mathcal{S}$.*

PROOF. The left direction of the proposition is trivial, as a total faithful assignment is also a partial assignment. For the right direction, from a partial and faithful assignment σ_0 for $\mathcal{S}$ and $\mathcal{G}$, we will construct a total and faithful assignment σ'.

To this end, we need to revise the *immediate evaluation operator* $\mathbf{T}$, introduced in (Corman et al., 2018a). As to update its definition to the notation used in this work, $\Sigma^{\mathcal{G},\mathcal{S}}$ is the set of all possible shape assignments for $\mathcal{S}$ and $\mathcal{G}$. Then, the immediate evaluation operator maps an assignment to another $\mathbf{T}(\sigma) : \Sigma^{\mathcal{G},\mathcal{S}} \to \Sigma^{\mathcal{G},\mathcal{S}}$ as follows:

$$\mathbf{T}(\sigma) = \{s(v) \mid [\text{def}(s)]^{v,\mathcal{G},\sigma} = 1\} \cup \{\neg s(v) \mid [\text{def}(s)]^{v,\mathcal{G},\sigma} = 0\}$$

Some properties can be shown for the $\mathbf{T}$ operator:

(i) First, if σ is faithful, then for every atom $s(v)$ or $\neg s(v)$ in σ it holds that $[\text{def}(s)]^{\mathcal{G},v,\sigma} = 1$ or 0 respectively, and therefore by definition should also be in $\mathbf{T}(\sigma)$. Therefore, if σ is faithful, then $\sigma \subseteq \mathbf{T}(\sigma)$.

(ii) All newly introduced atoms in $\mathbf{T}(\sigma)$ follow the defined semantics for function $[\phi]^{v,\mathcal{G},\sigma}$, so if σ is faithful, then $\mathbf{T}(\sigma)$ is also faithful.

(iii) For any $\sigma_1, \sigma_2 \in \Sigma_{\mathcal{G},\mathcal{S}}$, if $\sigma_1 \subseteq \sigma_2$, then any element $s(v)$ or $\neg s(v)$ in $\mathbf{T}(\sigma_1)$ it holds that $[\mathrm{def}(s)]^{\mathcal{G},v,\sigma} = 1$ or 0 respectively by definition. Since σ_2 only extends on the content of σ_1 and this cannot change the evaluation of the same constraints, then it also holds that the element should be in $\mathbf{T}(\sigma_2)$. Therefore, if $\sigma_1 \subseteq \sigma_2$ then $\mathbf{T}(\sigma_1) \subseteq \mathbf{T}(\sigma_2)$, meaning $\mathbf{T}$ is monotone.

Now, consider the set of assignments $\Sigma_0 \subseteq \Sigma^{\mathcal{G},\mathcal{S}}$ that extend σ_0. Meaning, $\sigma \in \Sigma_0$ if and only if $\sigma_0 \subseteq \sigma$. Thanks to monotonicity, for all $\sigma \in \Sigma_0$, $\mathbf{T}(\sigma_0) \subseteq \mathbf{T}(\sigma)$. And because σ_0 is faithful, $\sigma_0 \subseteq \mathbf{T}(\sigma_0)$. Therefore $\sigma_0 \subseteq \mathbf{T}(\sigma)$, meaning $\mathbf{T}(\sigma) \in \Sigma_0$ for all $\sigma \in \Sigma_0$. The latter, in addition with the fact that $\mathbf{T}$ is monotone over $\langle \Sigma_0, \subseteq \rangle$, by the weaker version of the Knaster-Tarski Theorem, $\mathbf{T}$ admits a fixed-point σ_1 over Σ_0. Because $\sigma_0 \subseteq \sigma_1$ and σ_0 is faithful, then also is σ_1.

The assignment σ_1 can be computed by recursively applying $\mathbf{T}$ starting with σ_0 until a fixed-point is reached. Now, σ_1 is not necessarily a total assignment, i.e. a pair of shape s and node v can exist such that neither $s(v)$ or $\neg s(v)$ is in σ_1. If that is the case, then the corresponding shape s cannot be terminal. This holds because it can be shown that assignments for terminal shapes should be computed for all nodes when applying $\mathbf{T}$ recursively, and therefore is included in the fixed point σ_1. Because s is not terminal, then $\mathrm{def}(s)$ must include a reference to another shape (or to itself) that is also not terminal.

Consider the set $N(\sigma_1)$ of all atoms $s(v)$ for shapes in $\mathcal{S}$ and nodes in $\mathcal{G}$, such that neither $s(v)$ or $\neg s(v)$ is in σ_1. Then, σ_1 can be further extended into σ', by the following definition, by considering the labeling function τ that makes $\mathcal{S}$ consistent:

$$\begin{aligned}\sigma' = \sigma_1 \\ \cup \{s(v) \mid s(v) \in N(\sigma_1) \text{ and } \tau(s) = 1\} \\ \cup \{\neg s(v) \mid s(v) \in N(\sigma_1) \text{ and } \tau(s) = 0\}\end{aligned}$$

First, note that σ' is total, since all non-terminal options are covered for every atom that is not covered by σ_1.

We will show that σ' is faithful, by analyzing case by case over its items. For every atom $s(v)$ or $\neg s(v)$ in σ' such that its also in σ_1, we already showed that the latter is faithful and therefore the corresponding constraint holds. Now, consider atoms that are not in σ_1:

- $s(v) \in \sigma'$ such that $s(v) \in N(\sigma_1)$ and $\mathrm{def}(s) = \neg s'$. In this case, $\tau(s) = 1$ (by definition of N), and $\tau(s') = 0$ (because of consistency). Neither $s'(v)$ or $\neg s'(v)$ can be in σ_1, if so then $s(v)$ or $\neg s(v)$ should had been in the fixed point of $\mathbf{T}$ by definition. Since $\tau(s') = 0$, then $\neg s'(v) \in \sigma'$ by definition, and therefore $[\mathrm{def}(s)]^{v,\mathcal{G},\sigma'} = 1$.
- $s(v) \in \sigma'$ such that $s(v) \in N(\sigma_1)$ and $\mathrm{def}(s) \neq \neg s'$. Since $\mathcal{S}$ is in normal form, all mentions of other shapes $s', s'', \ldots$ in $\mathrm{def}(s)$ must all be positive. Since neither $s(v)$ or its negation are part of σ_1 originally, then its because of the absence of other atoms for $s', s'', \ldots$ in σ_1 too. Since all mentions are positive and $\tau(s) = 1$, then $\tau(s') = \tau(s'') = \cdots = 1$ because of consistency. Therefore all unmentioned atoms for $s', s'', \ldots$ are included as positive atoms, which can only produce $[\mathrm{def}(s)]^{v,\mathcal{G},\sigma'} = 1$ since there is no negation involved.
- $\neg s(v) \in \sigma'$ such that $s(v) \in N(\sigma_1)$ and $\mathrm{def}(s) = \neg s'$. Similarly as the first case, neither $s'(v)$ or $\neg s'(v)$ can be in σ_1, but now $\tau(s) = 0$ and $\tau(s') = 1$. Then $s'(v) \in \sigma'$ by definition, and therefore $[\mathrm{def}(s)]^{v,\mathcal{G},\sigma'} = 0$.
- $\neg s(v) \in \sigma'$ such that $s(v) \in N(\sigma_1)$ and $\mathrm{def}(s) \neq \neg s'$. Similarly as the second case, but now $\tau(s) = \tau(s') = \tau(s'') = \cdots = 0$ with all positive mentions in $\mathrm{def}(s)$. All unmentioned atoms for $s', s'', \ldots$ are included as negative atoms, which can only produce $[\mathrm{def}(s)]^{v,\mathcal{G},\sigma'} = 0$.

Therefore, for all cases, if $s(v) \in \sigma'$ then $[\mathrm{def}(s)]^{v,\mathcal{G},\sigma'} = 1$; and if $\neg s(v) \in \sigma'$ then $[\mathrm{def}(s)]^{v,\mathcal{G},\sigma'} = 0$. Since σ_0 was faithful, and $\sigma_0 \subseteq \sigma'$, then all target constraints were already satisfied. Thus, σ' is a faithful and total assignment for $\mathcal{S}$ and $\mathcal{G}$.

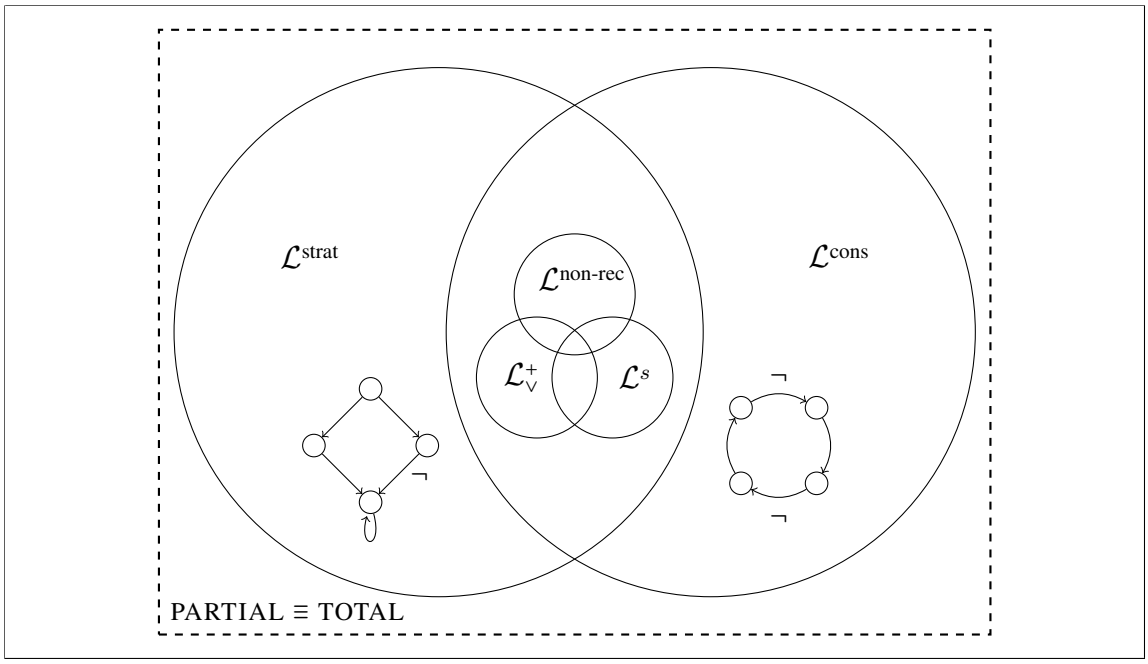

FIGURE 6.3. SHACL fragments hierarchy.

Therefore this proves that the partial and total validation problems coincide for $\mathcal{L}^{\text{cons}}$.

□

Then, although consistent SHACL and stratified SHACL form non-inclusive families, both of them are part of a bigger family of schemas if we consider the coincidence of partial and total validation problems as a fragment. Figure 6.3 shows the known SHACL fragments with the inclusion of the newly proposed $\mathcal{L}^{\text{cons}}$.

Nonetheless, tractability is not guaranteed for every schema in this bigger fragment, as we already showed that stratified SHACL can become NP-hard in data complexity. The good news is that for the case of consistent SHACL, the validation problem is tractable, which is proven in the next subsection.

6.2. Tractable algorithm

The fragment definition and the corresponding validation algorithm were both conceived by analyzing formulas from rule pattern evaluation explained in Section 5. The algorithm simply takes the resulting formula set for a schema and graph and applies logic resolution in an iterative manner until it cannot continue. Consistent schemas ensure that this unit propagation resolution fulfills certain properties that guarantee tractability.

Firstly, consistent schemas ensure that the generated formulas can fit a certain form by an appropriate renaming of propositional variables.

Consistent validation formula set. Consider consistent schema $\mathcal{S}$, and function τ as its corresponding labeling function. The *consistent validation formula set* $\overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau}$ is defined as the set of formulas, originally from the validation formula set $\Gamma_{\mathcal{G},\mathcal{S}}$, after renaming its propositional variables and formulas as follows. For every shape $s \in S$ such that $\tau(s) = 0$, then for every $s(v)$ variable present in $\Gamma_{\mathcal{G},\mathcal{S}}$ its corresponding negation variable $\overline{s}(v) = \neg s(v)$ is introduced and extends the variable set. Then, for every shape $s \in S$ such that $\tau(s) = 0$ and every formula f in $\Gamma_{\mathcal{G},\mathcal{S}}$ with mentions of $s(v)$ or $\neg s(v)$ is renamed by replacing $\neg\overline{s}(v)$ and $\overline{s}(v)$ respectively. Algorithm 3 shows this process, as it reassigns literals depending on its corresponding label. It also receives a boolean argument r that specifies in which way the renaming will be done, so that the same algorithm can be used to turn the renaming back to the original set of variables.

It is clear that $\overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau}$ is equivalent to $\Gamma_{\mathcal{G},\mathcal{S}}$ when considering the equivalence between replaced variables $\overline{s}(v) = \neg s(v)$. On the other hand, the appearance of renamed variables in formulas is very structured thanks to the schema being consistent and in normal form. By definition terminal shapes only have terminal shapes mentions in their corresponding rule pattern, so no formula that has a terminal shape atom as a head has any mention of renamed variables. Then renamed atoms may appear only in non-terminal shape headed formulas or as literals. Furthermore, formulas from non-terminal shapes involve at least one atom of non-terminal shapes with the same label value from τ. The only exceptions to the latter are formulas for negation constraints $\mathrm{def}(s) = \neg s'$, which by definition are the

Algorithm 3 RENAME ALGORITHM

Require: Schema $\mathcal{S} = \langle S, \text{targ}, \text{def} \rangle$, labeling function τ set F of formulas, and boolean argument r.

for all $f \in F$ **do**
  for all $s \in S$ where $\tau(s) = 0$ **do**
    if $r = 1$ **then**
      Replace $s(v)$ in f for $\neg\overline{s}(v)$
      Replace $\neg s(v)$ in f for $\overline{s}(v)$
    else
      Replace $\neg\overline{s}(v)$ in f for $s(v)$
      Replace $\overline{s}(v)$ in f for $\neg s(v)$
    end if
  end for
end for return F

only formulas that involve renamed and non-renamed atoms between only non-terminal shapes.

This shows that the renaming is only needed for a subset of formulas. We define the *terminal partition* $\mathcal{P}_{\mathcal{G},\mathcal{S},\tau} = (\mathcal{T}, \mathcal{C})$ of $\overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau}$ as the formula partition that takes into account the type of shape that generated it. We say a shape s *generated* a formula f if either f or its head is a literal for s and some node v, or a renamed version of one.

- $\mathcal{T} = \{f \in \overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau} \mid f \text{ is generated from shape } s \text{ such that } \tau(s) = 1/2\}$
- $\mathcal{C} = \{f \in \overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau} \mid f \text{ is generated from shape } s \text{ such that } \tau(s) \neq 1/2\}$

Now, we define three algorithms designed to carry out the logical resolution over a set of implication formulas in order to generate an proper assignment or proving that the set can not be satisfied. In these algorithms, we represent every formula as $C \rightarrow C_1 \vee C_2 \vee \ldots \vee C_\ell$, since this form adjust to every type of formula from the validation formula set. C and every C_i represent conjunctive clauses, and are treated as sets of the literals that compose the clause, or as $\top$ or $\bot$ for readability. RESOLUTION ALGORITHM is the main loop that performs unit propagation resolution over a set of implication formulas F using literals from a set σ. Then, in an iterative manner, it updates a set of literals known to be true and infers from implication formulas new literals that are satisfied. To achieve

this, it calls the other two procedures: REDUCE and INFER. The REDUCE ALGORITHM filters out any formulas that already are satisfied by the given literals; while the INFER ALGORITHM removes non-satisfied atoms out of formulas, which may potentially derive new literals to be introduced.

Algorithm 4 RESOLUTION ALGORITHM

Require: Set F of implication formulas and set σ of facts.

if $\{s(v) \mid s(v) \in \sigma\} \cap \{s(v) \mid \neg s(v) \in \sigma\} \neq \varnothing$ **then**
    $\sigma \leftarrow \sigma \cup \{\bot\}$
    return F, σ
end if
$F' \leftarrow F$
$\sigma' \leftarrow \sigma$
repeat
    $F'' \leftarrow F'$
    $\sigma'' \leftarrow \sigma'$
    $F' \leftarrow \text{REDUCE}(F'', \sigma'')$
    $F', \sigma' \leftarrow \text{INFER}(F', \sigma'')$
until $\bot \in \sigma' \ \vee \ F' = F''$
return F', σ'

Algorithm 5 REDUCE ALGORITHM

Require: Set F of implication formulas and set σ of facts.

$F' \leftarrow F$
for all $f = C \rightarrow C_1 \vee C_2 \vee \ldots C_\ell \in F$ **do**
    if $\sigma \cap \{\neg l \mid l \in C\} \neq \varnothing$ **then**
        $F' \leftarrow F' - \{f\}$
    end if
    for all $i, \ 1 \leq i \leq \ell$ **do**
        if $C_i \subseteq \sigma$ **then**
            $F' \leftarrow F' - \{f\}$
        end if
    end for
end for return F'

First, as we claim this is a method of logical resolution, is imperative that these procedures are correct, i.e. they produce equivalent sets of formulas, which is shown in the next proposition.

Algorithm 6 INFER ALGORITHM

Require: Set F of implication formulas and set σ of facts.

$F' \leftarrow \varnothing$
$\sigma' \leftarrow \sigma$
for all $f = C \to C_1 \vee C_2 \vee \ldots C_\ell \in F$ **do**
    $C' \leftarrow \{l \in C \mid l \notin \sigma\}$
    if $|C'| = 0$ **then**
        $C' \leftarrow \top$
    end if
    for all $i,\ 1 \leq i \leq \ell$ **do**
        if $\sigma \cap \{\neg l \mid l \in C_i\} \neq \varnothing$ **then**
            $C'_i \leftarrow \bot$
        else
            $C'_i \leftarrow \{l \in C \mid l \notin \sigma\}$
        end if
    end for
    $C'' \leftarrow \{C'_i \mid C_i \neq \bot\}$
    if $|C''| = 0$ **then**
        $f' \leftarrow C' \to \bot$
    else
        $f' \leftarrow C' \to \bigvee C''$
    end if
    if $f' = \top \to \bot$ **then**
        $\sigma' \leftarrow \sigma' \cup \{\bot\}$
    else if $f' = \top \to C'_1$ **then**
        $\sigma' \leftarrow \sigma' \cup \{l \mid l \in C'_1\}$
    else if $f' = C' \to \bot \ \wedge\ |C'| = 1$ **then**
        $\sigma' \leftarrow \sigma' \cup \{\neg l \mid l \in C'\}$
    else
        $F' \leftarrow F' \cup \{f'\}$
    end if
end for
return F', σ'

Proposition 6.4. *If F' and σ' are the result that* RESOLUTION(F, σ) *produces, then the sets of formulas $F \cup \sigma$ and $F' \cup \sigma'$ are equivalent.*

PROOF. First, note that sets σ and σ' contain only literals, while F and F' contain non-literal implication formulas. This proof will be separated in two, by showing equivalence of these formula sets first between the input and output of REDUCE algorithm, and then the same for the INFER algorithm.

In REDUCE, the compared sets are $F \cup \sigma$ and $F' \cup \sigma$, where F is the input set of non-literal formulas, and F' the output. Since F' is result of removing formulas from F, then $F' \subseteq F$ holds, and therefore $F \cup \sigma \vDash F' \cup \sigma$. Now, let β be a propositional assignment such that $\beta \vDash F' \cup \sigma$. It is direct that $\beta(f) = 1$ for all common formulas $f \in (F \cap F') \cup \sigma$. Then, what is left to show is that any formula $f = C \to C_1 \vee \ldots C_\ell \in F - F'$ also must be satisfied. If f was filtered out, then there was a reason for it and it was done either at line 4 or 8 in REDUCE.

- If it was at line 4, then $\sigma \cap \{\neg l \mid l \in C\} \neq \varnothing$. Therefore literal l exists in C such that $\beta(\neg l) = 1$ (by also being in σ), meaning $\beta(l) = 0$, and $\beta(C) = 0$. Since f is an implication, $\beta(f) = 1$.
- While if it was at line 8, then a clause C_i exists such that $C_i \subseteq \sigma$. Therefore meaning that $\beta(C_i) = 1 = \beta(C_1 \vee \ldots C_\ell) = \beta(f)$.

Thus, $F' \cup \sigma \vDash F \cup \sigma$, meaning $F \cup \sigma$ and $F' \cup \sigma$ are equivalent.

Now for INFER, the compared sets are $F \cup \sigma$ and $F' \cup \sigma'$. Since σ' is result of adding literals to σ, then $\sigma \subseteq \sigma'$ holds. Every iteration of the algorithm takes a formula $f \in F$, and may alter it and then add it to either F' or add literals to σ'. For any iteration, let $f = C \to C_1 \vee \ldots \vee C_\ell$ be the formula at the beginning of the iteration, and $f' = C' \to C'_1 \vee \ldots \vee C'_m$ be its value right after line 20.

First, we will show that for any assignment β that satisfies σ, then $\beta(f) = 1 \leftrightarrow \beta(f') = 1$. By cases for the right direction, if $\beta(f) = 1$ and $\beta(C) = 0$, then for at least one literal $l \in C$, $\beta(l) = 0$. Since β satisfies σ then l cannot be in it. This means $l \in C'$ (beacuse of line 4), and therefore $\beta(C') = 0$, while $\beta(f') = 1$. The other case for $\beta(f) = 1$ is when $\beta(C_1 \vee \ldots \vee C_n) = 1$. This means there is a clause C_i such that

$\beta(C_i) = 1$, where all literals l in it are valuated to true ($\beta(l) = 1$). For the satisfied clause C_i, an altered clause C'_j is defined in lines 9 through 12. Since no negated literal $\neg l$ can be in σ with $\beta(\neg l) = 0$, then $C'_j \neq \bot$ and therefore $C'_j \subseteq C_i$. Thus the valuation for f' is met: $\beta(C'_j) = 1 = \beta(C'_1 \vee \ldots \vee C'_m) = \beta(f')$. For the left direction we will prove the contrapositive of the right direction. If $\beta(f) = 0$, then the one possible valuation is $\beta(C) = 1$ and $\beta(C_1 \vee \ldots \vee C_n) = 0$. If so, then it hold for all literals l from clause C that $\beta(l) = 1$, and since $C' \subseteq C$ then $\beta(C') = 1$. Now, since $\beta(C_1 \vee \ldots \vee C_n) = 0$, then for every C_i there is a literal l_i in it such that $\beta(l_i) = 0$. Clauses in $C'_1 \vee \ldots \vee C'_m$ by construction (line 12) are such that there is a corresponding C_i in f where $C'_j = \{l \in C \mid l \notin \sigma\}$. Since the mentioned literals l_i satisfy $\beta(l_i) = 0$, then l_i are not contained in σ, and therefore $l_i \in C'_j$. Then, every clause in $C'_1 \vee \ldots \vee C'_m$ contains a literal l such that $\beta(l) = 0 = \beta(C'_j)$, and therefore $\beta(C'_1 \vee \ldots \vee C'_m) = 0$. Thus $\beta(f') = 0$, because $\beta(C'_1 \vee \ldots \vee C'_m) = 0$ and $\beta(C') = 1$.

What is left to prove is that $F \cup \sigma$ and $F' \cup \sigma'$ are equivalent in INFER. First the right direction, for any assignment β that $\beta \vDash F \cup \sigma$, then $\beta \vDash \sigma$ and $\beta(f) = 1$ for all $f \in F$. Thanks to the latter property shown, then for all altered formulas f' it also holds that $\beta(f') = 1$. Since $\beta(f') = 1$, then one of the options between lines 23 and 28 added formulas into either σ' or F'. If $f' = \top \to C'_1$, then $\beta(C'_1) = 1$ as for all literals $l \in C'_1$: $\beta(l) = 1$. If $f' = C' \to \bot$ and $|C'| = 1$, then $\beta(C') = 0$ as for the only literal $l \in C'$, therefore $\beta(\neg l) = 1$. If neither of the previous cases hold, then f' is added which also holds $\beta(f') = 1$. Then, for any case of added formula into σ' or F' it holds that β assigns to 1, then $\beta \vDash F' \cup \sigma'$. For the left direction, for any assignment β that $\beta \vDash F' \cup \sigma'$, then $\beta \vDash \sigma'$, and since $\sigma \subseteq \sigma'$ then $\beta \vDash \sigma$. Cases left to prove are for formulas $f \in F$. These cases apply the same argument as the previous direction, since all iterations of INFER transform a formula f' for each $f \in F$ that is equivalent for assignments β' such that $\beta' \vDash \sigma$. Then, depending on the form of the resulting formula f', formulas are added into $F' \cup \sigma'$, and since all of these have $\beta(f') = 1$, then the corresponding initial formula f also $\beta(f) = 1$.

Finally, since the input and output formula sets in REDUCE and in INFER are equivalent, then so are the initial and final formula sets at each iteration of RESOLUTION.

□

Equivalence holds and the procedure does so by reducing the atoms in each implication formula by the presence of the expanding set of literals. This procedure then simulates the satisfaction checking of shape constraints by only focusing on shape assignment dependencies. Conditions concerning locality of a node are previously checked and filtered by the corresponding queries that then gave origin to these formulas.

As discussed before, the revision of constraint satisfaction is easy for terminal shapes, since they behave as non-recursive schemas. In the context of the RESOLUTION algorithm, the following proposition shows that formulas for terminal shapes always completely resolve.

Proposition 6.5. *Let $\mathcal{S}$ be an consistent shape schema, τ the corresponding labeling function, $\mathcal{G}$ a graph and $\mathcal{P}_{\mathcal{G},\mathcal{S},\tau} = (\mathcal{T}, \mathcal{C})$ the terminal partition of the consistent validation formula set. Consider the execution of* RESOLUTION(F, σ) *where F is the set of all implication formulas in $\mathcal{T}$ and σ is the set of literals in $\mathcal{T}$. If the algorithm returns σ' and F', and $\bot \notin \sigma'$, then $F' = \varnothing$.*

PROOF. Firstly, all formulas considered come from rule patterns for terminal shapes, atomic and non-atomic, none have renamed literals, and all mentioned literals also come from terminal shapes, by definition. Then, none of this shapes are part of a cycle in the dependency graph $G_{\mathcal{S}}$, and all of their rule patterns either do not depend on other shapes, or depend only of other terminal shapes.

Let's assume RESOLUTION(F, σ) returns F' and σ' such that $\bot \notin \sigma'$ and $F' \neq \varnothing$. Then, at least one implication formula $f \in F'$ exists at the end of the procedure. Since $\bot \notin \sigma'$, the only way for the main loop to stop is if in one iteration the condition $F' = F''$ holds. Therefore, at the last iteration all implication formulas in F'' at the beginning are also at the end in F'.

This includes f, it may be an original formula from $\mathcal{T}$, or a simplified formula derived in the INFER procedure. But, it must come from either a rule or a complement formula

for a terminal shape s. At the same time, s must depend on other shapes for this to happen, since sink nodes only produce simple rule patterns as $\top \rightarrow s(v)$, which is derived into atomic formulas in INFER (line 23 and 24). Then, f is an implication with at least two literals l_1, l_2, for different terminal shapes. Since f was not filtered out or simplified in either the REDUCE or INFER procedures, this means neither of those literals or their negations are part of the atomic formulas derived: $\{l_1, l_2, \neg l_1, \neg l_2\} \cap \sigma'' = \varnothing$. Thanks to the original $\Gamma_{\mathcal{G},\mathcal{S}}$ formulas construction, another appearance for either literal must exists in another implication formula $f' \in F''$. The same argument can be applied over f', implying the existence of a third formula f'', as for the latter, and so on. If we apply this argument enough times and use the the pigeonhole principle, this results in a necessary cycle of dependencies between terminal shapes.

Therefore, our initial assumption arrived as to a contradiction over the definition of terminal shapes. Implying that if $\bot \notin \sigma'$, then $F' = \varnothing$, and thus proving the proposition.

□

Now, Proposition 6.6 shows a more interesting result about the resolution evaluation for non-terminal shape formulas. Consistency guarantees that any final set of formulas left in the algorithm can be trivially satisfied by assigning value true to every left variables in the set. This property is the key aspect that was identified for consistent schemas, and is what ensures tractability of the general validation algorithm.

Proposition 6.6. *Let $\mathcal{S}$ be an consistent shape schema, τ its corresponding labeling function, $\mathcal{G}$ a graph and $\overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau}$ the corresponding consistent validation formula set. Consider the execution of* RESOLUTION(F, σ) *where F is the set of all implication formulas in $\overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau}$ and σ is the set of literals in $\overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau}$. If the algorithm returns σ' and F', and $\bot \notin \sigma'$, then F' can be trivially satisfied.*

PROOF. Firstly, note that if $\bot \notin \sigma'$ then Proposition 6.5 holds when considering formulas that correspond to terminal shapes, therefore all formulas left in F', if any, must correspond to non-terminal shapes. The inclusion of non-terminal shape formulas do not

affect that result, because terminal shapes can resolve completely by themselves. We take this detail into account since it will simplify the proposed proof if we ignore terminal shape formulas all together.

Specifically, we can consider the execution of RESOLUTION(F, σ) as equivalent to doing it in a two step process: first execute RESOLUTION(F_1, σ_1) using only the terminal shape formulas F_1 and literals σ_1, which produces the literal set $\sigma_{\mathcal{T}}$; and then second execute RESOLUTION$(F_2, \sigma_2 \cup \sigma_{\mathcal{T}})$ using the non-terminal shape formulas F_2 and literals σ_2 in addition to the resulting literals $\sigma_{\mathcal{T}}$ from the previous execution.

Then, to prove the statement we will focus on the second execution where terminal formulas are already resolved. First, we will show that the initial set of formulas can be trivially satisfied by assigning value true to every left variables, and then that during execution of RESOLUTION this property does not change.

The initial set of formulas F are taken from $\mathcal{C}$, and since they come from non-terminal shapes, all rule patterns depend on other shapes. And on the other hand, since variable renaming was applied to non-terminal shapes, renamed variables may be present in this set. But, thanks to the proper renaming based on consistency and the schema being in normal form, each possible formula adjust to a restricted form that can be satisfied by assigning 1 to every propositional variable present in this set. Let's see by cases, all possible formulas that start initially in F:

- rule formulas for not renamed shapes present all positive literals at both sides of the implication, either as $s_1(v_1) \wedge \ldots \wedge s_n(v_n) \rightarrow s(v)$ or $\overline{s'}(v) \rightarrow s(v)$. This holds because negation is only possible as single shape definition, and $\mathcal{S}$ is in normal form. For all other cases, not renamed shapes have in their constraint definition other not renamed shapes, since $\mathcal{S}$ is also consistent. Both types of formulas can be satisfied by assigning its head $s(v)$ to 1.
- complement formulas from rules for not renamed shapes are either $s(v) \rightarrow C_1 \vee \ldots C_\ell$ or $s(v) \rightarrow \overline{s'}(v)$, where $C_i = s_{i,1}(v_{i,1}) \wedge \ldots \wedge s_{i,n}(v_{i,n})$. As the previous case, these formulas present all positive literals at both sides of the implication, for the same argument and

by construction of complement formulas. But in this case, by assigning 1 to all clauses in the body (right side) of the implications the whole formula is satisfied.

- rule formulas for renamed shapes present all negative literals at both sides of the implication, either as: $\neg s'(v) \to \neg\overline{s}(v)$; or $l_1 \wedge \ldots \wedge l_n \to \neg\overline{s}(v)$, where at least one literal l_i is of the form $\neg\overline{s'}(v')$. The latter follows from the fact that if s had its atoms renamed, then $\tau(s) = 0$ and therefore depends of other non-terminal shapes such that $\tau(s') = 0$. If any literal l_j is positive, then that is because $\tau(s') = 1/2$ and $\mathcal{S}$ is in normal form and is consistent. This case can be satisfied by assigning 1 to the variables in the body of these formulas, which values to 0 the left side of the implication, thus values to 1 the whole formula.
- complement formulas from rules for renamed shapes are either $\neg\overline{s}(v) \to C_1 \vee \ldots \vee C_\ell$ or $\neg\overline{s}(v) \to \neg s'(v)$, where C_i are conjunctions clauses of literals where at least one for each is of the form $\neg\overline{s'}(v')$. These form follows from same argument as the previous cases. Now, by assigning 1 to the variable in the head of the formula (left side) the implications as a whole is satisfied, since the left side values to 0.

Thus, this shows that the proper renaming of variables and the consistency of the schema produces a formula set that can be trivially satisfied by assigning 1 to all present variables in the set.

Now, this might change during execution of RESOLUTION when taking into account the set of literals σ. To show that the trivial satisfaction is not lost during execution, we will prove the following property over the iterations of the main loop on RESOLUTION: At any given iteration of its main loop, **neither** of the following happen:

- heads of rule formulas, that come from not renamed shapes, are simplified into $\bot$
- heads of complement formulas, that come from renamed shapes, are simplified into $\top$

This will be proven by induction over the number of iterations the main loop of RESOLUTION goes through. First, we will consider the base case, i.e. the first iteration, and then we'll show the inductive case.

Base case. The initial set F' is taken from $\mathcal{C}$, whose formulas were in the form that was already described, and σ' is taken from $\mathcal{C}$ and literals for terminal shapes. Now, all facts in σ' come from either:

- target formulas, which only results in positive atoms $s(v)$, since $\mathcal{S}$ is consistent;
- non-retrieval formulas for not renamed shapes, resulting in negative atoms $\neg s(v)$;
- non-retrieval formulas for renamed shapes, resulting in positive atoms $\overline{s}(v)$; and
- literals of terminal shapes, that may be positive or negative.

In iteration 1 of RESOLUTION, the current formula sets F'', σ'' start as previously mentioned. Once the REDUCE subroutine filters out any satisfied formulas forming F', the INFER subroutine may simplify any of the remaining non-atomic formulas in F'. Let's say the stated property does not hold and the head of a formula f (of the mentioned cases) was indeed simplified:

- For the first case, if f is a rule formula for a not renamed shape s and $s(v)$ is its head that was simplified into $\bot$. If that is the case, the formula may be of either forms: $s_1(v_1) \wedge \ldots \wedge s_n(v_n) \rightarrow s(v)$ or $\overline{s'}(v) \rightarrow s(v)$. Either way, in line 17 of INFER the head got simplified, because $\sigma \cap \{\neg l \mid l \in \{s(v)\}\} \neq \varnothing$ (line 9 of INFER). This directly means the negated atom $\neg s(v)$ was in σ initially. As stated before, the only negative atoms for not renamed (and non-terminal) shapes are non-retrieval formulas, meaning $\neg s(v)$ was a non-retrieval formula for s. Since the evaluation of rule patterns can not produce both a non-retrieval formula and a rule formula f for the same node v, a contradiction is reached.
- Now, let's consider f is an complement formula for renamed shape $\overline{s}$ and $\overline{s}(v)$ is its head that was simplified into $\top$. If that is the case, the formula may be of either forms: $\neg\overline{s}(v) \rightarrow C_1 \vee \ldots C_\ell$ or $\neg\overline{s}(v) \rightarrow \neg s'(v)$, Either way, in line 6 of INFER the head got simplified, because $|\{l \in \{\neg\overline{s}(v)\} \mid l \notin \sigma\}| = 0$ (line 4 of INFER). This directly means the atom $\neg\overline{s}(v)$ was in σ initially. This cannot be, since as shown before, there are no

negative atoms in σ for renamed shapes at the start. Again, we arrive to a contradiction for the existence of f.

Therefore the existence of simplified f cannot be, and therefore the property holds in the base case. Another important aspect to consider, is that after this iteration there are no atoms for terminal shapes left in implication formulas thanks to the REDUCE and INFER procedures.

Induction step. Let's assume that the proposition holds for all iterations up to number $i - 1$. No heads of the mentioned types of formulas have been simplified yet.

Let's assume that in iteration i that the head of a formula f (of the mentioned types) is indeed simplified into a formula f'. Let's consider which type of formula is f by cases:

- Let f be a rule formula for not renamed shape s and $s(v)$ is its head that was simplified into $\bot$. As shown before, this occurs because of the presence of $\neg s(v)$ in σ, but $\neg s(v)$ can not in the initial fact set σ. Therefore it appeared in later iteration k before iteration i. The only way for a new formula or fact to appear in this set is through the INFER subroutine, by simplifying another a formula f''. There are four possible types of formulas for f'' that could be a source for $\neg s(v)$ as a derived result:
  - Suppose f'' was a complement formula of the form $s(v) \rightarrow C_1 \vee \ldots C_\ell$, where $C_i = s_{i,1}(v_{i,1}) \wedge \ldots \wedge s_{i,n}(v_{i,n})$. If $\neg s(v)$ was obtained in line 26 of INFER, then all clauses C_i where were simplified in line 9 and 10. This means that for each C_i: $\sigma \cap \{\neg s_{i,1}(v_{i,1}), \ldots, \neg s_{i,n}(v_{i,n})\} \neq \varnothing$ holds, meaning at least one negative literal of each clause was already in σ. Because of the construction of complement formulas $[\![p^{\leftarrow}_{\mathcal{S}}]\!]^{\mathcal{G}}$, each clause C_i comes from a rule formula that has that clause as part of its body, where one of those formulas must be f. Since every single one of those formulas has a negative literal in σ already, then every single one must have been filtered out in lines 3 and 4 of REDUCE before even reaching the INFER subroutine.

- If f'' was a complement formula of the form $s(v) \rightarrow \overline{s'}(v)$, the same argument as the previous case can be applied, such that f should had been filtered out before reaching the simplification point.
- Suppose f'' is an rule formula of a not renamed shape with $s(v)$ as part of the body. To obtain $\neg s(v)$ as result of the simplification in line 26 of INFER, then f'''s head had to be simplified in this iteration, that is previous to iteration i. But, by our induction hypothesis, all previous iterations had no simplification of heads for rule formulas of a not renamed shapes. So, this cannot be.
- Suppose f'' was an complement formula of a renamed shape $\overline{s'}$ such that $\mathrm{def}(s') \coloneqq \neg s$, which produces a formula of the form $\neg\overline{s'}(v) \rightarrow \neg s(v)$. Again, this would mean that the head of a complement formula of a renamed shape is simplified in a previous iteration to i, contradicting the hypothesis.

- Let f be an complement formula for a renamed shape $\overline{s}$ and $\overline{s}(v)$ is its head that was simplified into $\top$. As shown in the base case, this occurs because of the presence of $\neg\overline{s}(v)$ in σ, but $\neg\overline{s}(v)$ can not in the initial fact set σ. Therefore it appeared in later iteration k before iteration i. There are four possible types of formulas for f'' so that $\neg\overline{s}(v)$ is obtained from it:
  - Suppose f'' is an rule formula for the renamed shape $\overline{s}$ and $\neg\overline{s}(v)$ being its head: $\neg\overline{s_1}(v_1) \wedge \ldots \wedge \neg\overline{s_n}(v_n) \rightarrow \neg\overline{s}(v)$. $\neg\overline{s}(v)$ must have been obtained in line 24 of INFER, meaning all its negative literals in the body of the formula got filtered out in line 4. Therefore, all $\neg\overline{s_1}(v_1), \ldots, \neg\overline{s_n}(v_n)$ were in σ already. If that is the case, these literals conform one of the clauses C_i in f, because of the construction of $[\![p^{\leftarrow}_{\mathcal{S}}]\!]^{\mathcal{G}}$. Since the whole clause is already in σ for iteration k, the formula f should had been filtered out by the REDUCE (lines 7 and 8) routine by iteration i.
  - If f'' was a rule formula of the form $\neg s'(v) \rightarrow \neg\overline{s}(v)$, the same argument as the previous case can be applied, such that f should had been filtered out before reaching the simplification point.

- Suppose f'' is another complement formula with $\neg\overline{s}(v)$ as part of its body. Similarly as before, to obtain $\neg\overline{s}(v)$ as result of the simplification, the head of f'' had to be simplified in a previous iteration. This reaches a contradiction thanks to our induction hypothesis.
- Suppose f'' was a rule formula of a not renamed shape s' such that $\mathrm{def}(s) := \neg s'$, which produces a formula of the form $\overline{s}(v) \to s'(v)$. Again, this would mean that the head of a rule formula of a not renamed shape is simplified in a previous iteration to i, contradicting the hypothesis.

Therefore, all possibilities arrive to a contradiction, then the assumption of the existence of f cannot be. Thus, finally proving that the induction hypothesis holds and therefore proving the desired property.

By proving this, the only possibilities of formulas not being satisfied by the trivial assignment are gone. As it can be shown by every case of formulas that are kept in RESOLUTION, they always can be satisfied by that assignment, or are simplified into atoms:

- For rule formulas for not renamed shapes: $s_1(v_1) \wedge \ldots \wedge s_n(v_n) \to s(v)$ or $\overline{s'}(v) \to s(v)$. Since their head never gets simplified into $\bot$, they still can get satisfied by assigning its head $s(v)$ to 1.
- For complement formulas from rules for not renamed shapes: $s(v) \to C_1 \vee \ldots C_\ell$ or $s(v) \to \overline{s'}(v)$, where $C_i = s_{i,1}(v_{i,1}) \wedge \ldots \wedge s_{i,n}(v_{i,n})$. If the whole body gets simplified, the head gets added to σ and does not continue as an implication formula. While if the head gets simplified, the formula still can be satisfied by assigning 1 to all clauses in the body.
- For rule formulas for renamed shapes: $\neg\overline{s_1}(v_1) \wedge \ldots \wedge \neg\overline{s_n}(v_n) \to \neg\overline{s}(v)$ or $\neg s'(v) \to \neg\overline{s}(v)$. If the head gets simplified, the formula still can be satisfied by assigning 1 to the variables in the body of these formulas. While if the body gets simplified, the head gets added to σ and does not continue as an implication formula.

- complement formulas from rules for renamed shapes: $\neg\overline{s}(v) \rightarrow C_1 \vee \ldots C_\ell$ or $\neg\overline{s}(v) \rightarrow \neg s'(v)$, where $C_i = \neg\overline{s_{i,1}}(v_{i,1}) \wedge \ldots \wedge \neg\overline{s_{i,n}}(v_{i,n})$. Since their head never gets simplified into $\top$, these still can get satisfied by assigning $\overline{s}(v)$ to 1.

Summing it all up, the set of implication formulas left for non-terminal shapes can be trivially satisfied by assigning all left propositional variables in these formulas to 1. When passed through the execution of RESOLUTION with the set of literals, these implication formulas get either filtered out or simplified. But, during its execution no formulas are ever simplified into another that cannot be later satisfied trivially. Thus, the resulting set can also be trivially satisfied by assigning the remaining variables to 1.

□

Thanks to the proved propositions, a general validation algorithm can be defined for consistent SHACL and is represented in Algorithm 7. The key idea behind it is to take the consistent validation formula set, resolve the shape assignments that can be validated, and if a contradiction is not reached when resolution is terminated, then the graph is valid and all unresolved formulas are trivially satisfied by assigning 1 to pending variables (and therefore shape assignments).

Algorithm 7 VALIDATION ALGORITHM

Require: Consistent shape schema $\mathcal{S}$, corresponding labeling function τ and graph $\mathcal{G}$.

$\Gamma_{\mathcal{G},\mathcal{S}} \leftarrow [\![p_\mathcal{S}]\!]^\mathcal{G} \cup [\![p_\mathcal{S}^{\leftarrow}]\!]^\mathcal{G} \cup [\![t_\mathcal{S}]\!]^\mathcal{G} \cup [\![a_\mathcal{S}]\!]^\mathcal{G}$
$\overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau} \leftarrow \text{RENAME}(\mathcal{S}, \tau, \Gamma_{\mathcal{G},\mathcal{S}}, 1)$
$F \leftarrow \{f \in \overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau} \mid f \text{ is non-atomic}\}$
$\sigma \leftarrow \{f \in \overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau} \mid f \text{ is atomic}\}$
$F', \sigma' \leftarrow \text{RESOLUTION}(F, \sigma)$
if $\bot \in \sigma'$ **then return** `FALSE`
end if
if $F' \neq \varnothing$ **then**
    $\sigma' = \sigma' \cup \text{SATURATE}(F')$
end if
$\sigma' \leftarrow \text{RENAME}(\mathcal{S}, \tau, \sigma', 0)$
return σ'

Algorithm 7 calls the procedure SATURATE (depicted by Algorithm 8) that takes all pending literals left in formulas and adds them as positive literals. Which is then again renamed back to the original propositional variables that represent the original shape assignment.

Algorithm 8 SATURATE ALGORITHM

Require: Set of implication formulas F

1: $\sigma'' = \varnothing$
2: **for all** $f = C \rightarrow C_1 \vee C_2 \vee \ldots C_n \in F$ **do**
3: $\quad \sigma_+ \leftarrow \{l \mid l \in C \cup C_1 \cup \ldots \cup C_n$ and is positive$\}$
4: $\quad \sigma_- \leftarrow \{\neg l \mid l \in C \cup C_1 \cup \ldots \cup C_n$ and is negative$\}$
5: $\quad \sigma'' = \sigma'' \cup \sigma_+ \cup \sigma_-$
6: **end for**
7: **return** σ''

Theorem 6.1 sums up the correctness of the whole VALIDATION algorithm for consistent schemas, and Theorem 6.2 establishes tractability for the problem.

Theorem 6.1. *Consider a consistent shape schema $\mathcal{S}$, its corresponding labeling function τ and graph $\mathcal{G}$. $\mathcal{G}$ is valid against $\mathcal{S}$ if and only if* VALIDATION$(\mathcal{S}, \tau, \mathcal{G})$ *returns a faithful assignment for $\mathcal{G}$ and $\mathcal{S}$.*

PROOF. Proposition 6.3 showed that for consistent schemas the partial and total validation problems coincide, and thanks to (Corman et al., 2019) (Proposition 6), we know that $\mathcal{G}$ is valid against $\mathcal{S}$ with respect to total assignments if and only if $\Gamma_{\mathcal{G},\mathcal{S}}$ is satisfiable under boolean semantics. The consistent validation formula set $\overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau}$ is clearly equivalent to $\Gamma_{\mathcal{G},\mathcal{S}}$ since it replaces some variables with their negation. Therefore, $\mathcal{G}$ is valid against $\mathcal{S}$ if and only if $\overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau}$ is satisfiable under boolean semantics.

First we focus on the right direction. If $\mathcal{G}$ is valid against the consistent schema $\mathcal{S}$, then $\overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau}$ is satisfiable under boolean semantics. VALIDATION then separates this set into a partition $\sigma \cup F$ of atomic and non-atomic formulas that are used to call RESOLUTION in line 5. Proposition 6.4 showed that every iteration of this call will produce an equivalent set partition $\sigma' \cup F'$. Then, satisfiability also spreads to each partition $\sigma' \cup F'$ thanks to equivalence.

To continue, we define a shape assignment construction from a propositional assignment β as: $\Sigma(\beta) = \{s(v) \mid \beta(s(v)) = 1\} \cup \{\neg s(v) \mid \beta(s(v)) = 0\}$.

Satisfability and equivalence also translates into the fact that on every iteration in RESOLUTION, for every assignment β that satisfies $\sigma' \cup F'$, the assignment $\Sigma(\beta)$ is faithful for $\mathcal{G}$ and $\mathcal{S}$. Not only that, but $\sigma' \subseteq \Sigma(\beta)$ also holds. The initial value for $\sigma' \cup F'$ before the first iteration is equivalent to the original set of formulas $\Gamma_{\mathcal{G},\mathcal{S}}$. The proof for Proposition 6 in (Corman et al., 2019) showed that for an arbitrary boolean assignment β that satisfies $\Gamma_{\mathcal{G},\mathcal{S}}$, the constructed shape assignment $\Sigma(\beta)$ is faithful for $\mathcal{G}$ and $\mathcal{S}$. After that, every iteration of RESOLUTION only expands the set of atomic literals, and since equivalence is maintained, this set also must be a valid shape assignment that agrees with a faithful assignment for $\mathcal{G}$ and $\mathcal{S}$, i.e. $\sigma' \subseteq \Sigma(\beta)$.

Now, let $\sigma' \cup F'$ be the resulting partition after the whole RESOLUTION call. Proposition 6.5 indicates that no implication formulas for terminal shapes are left in F', while Proposition 6.6 indicates that F' can be trivially satisfied. Specifically, the assignment $\beta^+_{F'}$ that assigns 1 to every propositional variable left in F' satisfies this set. Consider $\beta^+_{\sigma'}$ as the assignment that assigns 1 to every literal in σ'. Since the graph is valid, then RESOLUTION stopped after an iteration did not alter any formulas left in F'. Then, a whole iteration went by where the final σ' did not change. Thanks to the call of the procedures REDUCE and INFER, none of the left formulas in F' at the end of any iteration have any mentions of literals or their negations that are in σ' at the beginning of the same iteration. Therefore, this guarantees that σ' and F' do not share any propositional variables, and therefore the mentioned assignments can easily be extended into $\beta^+ = \beta^+_{F'} \cup \beta^+_{\sigma'}$, such that $\beta^+ \vDash \sigma' \cup F'$. Therefore, $\Sigma(\beta^+)$ is a faithful shape assignment for $\mathcal{G}$ and $\mathcal{S}$. Since σ' only contains atomic formulas, its easy to see that $\Sigma(\beta^+) = \sigma' \cup \Sigma(\beta^+_{F'})$. $\Sigma(\beta^+_{F'})$ is indeed what the procedure call SATURATE generates in line 9.

Thus, RESOLUTION returns $\Sigma(\beta^+)$, which is a faithful assignment for $\mathcal{G}$ and $\mathcal{S}$.

The left direction of the proposition can be easily seen as its contrapositive. If $\mathcal{G}$ is not valid against $\mathcal{S}$, then $\overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau}$ is unsatisfiable. If that the case, the $\perp$ mark will appear in

the RESOLUTION iteration since it is equivalent to unit propagation resolution application. Note, that even though unit propagation resolution is not a complete refutation procedure in general, as shown in Proposition 6.6, since $\mathcal{S}$ is consistent and RESOLUTION stops without $\bot$, then the resulting set of formulas F' can always be satisfied. Therefore, if the whole set is unsatisfiable, the $\bot$ mark has to appear eventually. Therefore, RESOLUTION will stop and VALIDATION will return `FALSE` in line 6.

□

Theorem 6.2. *Consider a consistent shape schema $\mathcal{S}$, its corresponding labeling function τ and graph $\mathcal{G}$. The execution of* VALIDATION$(\mathcal{S}, \tau, \mathcal{G})$ *is tractable in data complexity.*

PROOF. Firstly, line 1 computes the original set of formulas $\Gamma_{\mathcal{G},\mathcal{S}}$ for rule patterns. Its size is in $\mathcal{O}\big(|\bigcup_{s \epsilon S}([\![q_{\mathrm{def}(s)}]\!]^{\mathcal{G}} \cup [\![\mathrm{targ}(s)]\!]^{\mathcal{G}})|\big)$. This holds because of construction of rule patterns and formulas, where $|[\![t_{\mathcal{S}}]\!]^{\mathcal{G}}| \in \mathcal{O}(|\bigcup_{s \epsilon S}[\![\mathrm{targ}(s)]\!]^{\mathcal{G}}|)$ and $|[\![p_{\mathcal{S}}]\!]^{\mathcal{G}}|+|[\![p^{\leftarrow}_{\mathcal{S}}]\!]^{\mathcal{G}}|+|[\![a_{\mathcal{S}}]\!]^{\mathcal{G}}| \in \mathcal{O}(|\bigcup_{s \epsilon S}[\![q_{\mathrm{def}(s)}]\!]^{\mathcal{G}}|)$. Hence, the size of the rules we need for inference is not directly dependent on the size of the graph, but rather on the amount of targets and tuples that the SHACL schema selects to be validated. Furthermore, $[\![q_{\mathrm{def}(s)}]\!]^{\mathcal{G}}$ and $[\![\mathrm{targ}(s)]\!]^{\mathcal{G}}$ can be computed in polynomial time in data-complexity (as shown in (Pérez et al., 2009)), when $\mathcal{S}$ is considered to be fixed, and thus the set of rules can be computed in polynomial time in data complexity.

Then line 2 renames some of the non-terminal shapes defined by the labeling, which takes linear time in the size of the original set when $\mathcal{S}$ is fixed. The resulting consistent validation formula set $\overline{\Gamma}_{\mathcal{G},\mathcal{S},\tau}$ has the same size as $\Gamma_{\mathcal{G},\mathcal{S}}$. Then, the atomic and non-atomic partition is created in lines 3 and 4, which again can be done in linear time in the size of the amount of shapes.

Line 5 calls the RESOLUTION procedure, which simulates unit propagation resolution over the set of formulas. Unit propagation resolution usually takes quadratic time over the size of the whole formula sets, considering the amount of literals involved in each formula. The amount of different literals that these formulas can contain is also in the

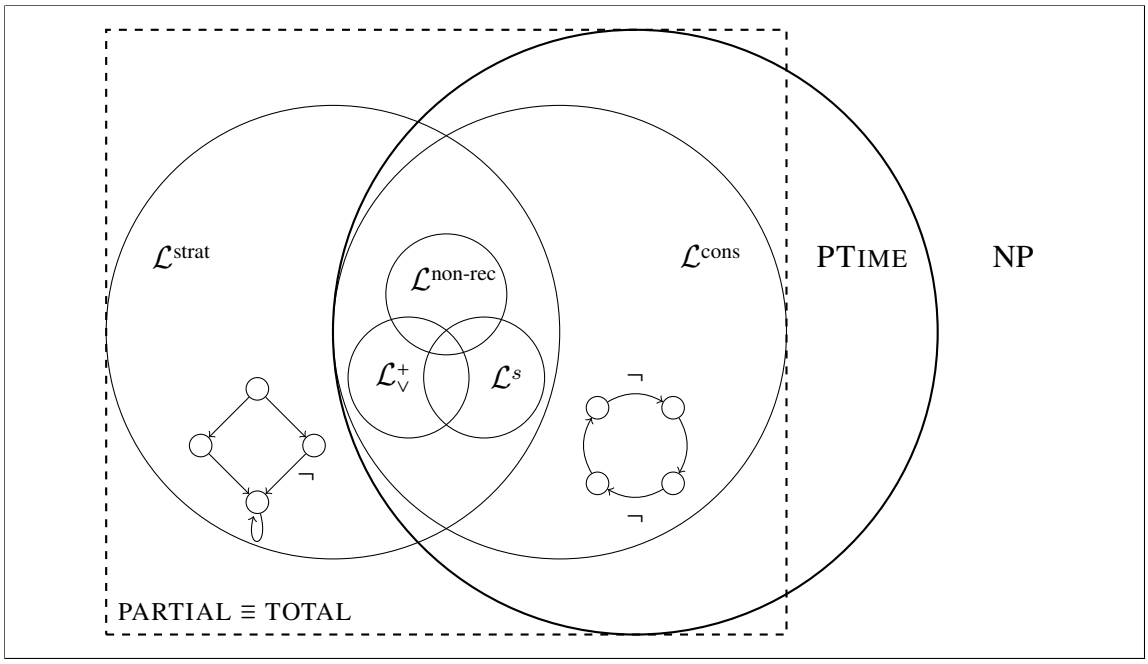

FIGURE 6.4. SHACL fragment hierarchy considering running data-complexity.

order of $\mathcal{O}\big(|\bigcup_{s\in S}([\![q_{\text{def}(s)}]\!]^{\mathcal{G}} \cup [\![\text{targ}(s)]\!]^{\mathcal{G}})|\big)$. The shown implementation for REDUCE and INFER may border the cubic time in that size, but it can easily be optimized with efficient data structures that keep track of the inclusion of literals for each formula, taking it down to almost linear in the size of the original set of formulas. Nonetheless, is still polynomial in the size of the set of formulas.

Lines 6 through 11 also can be done in linear time over the size of the set of formulas, for similar reasons as the previous points.

Thus, the whole VALIDATION procedure can be computed in polynomial time in data complexity.

□

This finally shows that the proposed fragment $\mathcal{L}^{\text{cons}}$ is indeed in PTIME in data complexity. Figure 6.4 summarizes the known SHACL hierarchy previously described, but considering the complexity classes for the corresponding validation problems.

7. CONCLUSIONS AND FUTURE WORK

First, we showed through different approximations that validation for non-recursive schemas can be treated in multiple ways, either by direct in-memory revision or through complete delegation of the task to SPARQL engines. These approaches brought different trade-offs to consider, and gave a base intuition on how to handle harder schemas.

Then, a review was made of the up to now known tractable and intractable SHACL fragments, and discussed the general challenges that each met. As well a revision of a validation approach that use both SPARQL and in-memory processing for the general recursive case was done. And using the latter points, we stated the main contribution of this thesis. Which was the proposal and definition of a SHACL fragment and tractable algoritm that could be applied for the revised non-recursive, all positive, strictly stratified schemas and even more schemas. Very interestingly, also introduced the first tractable fragment that included negation in cycles.

As for future work, there are some expressivity and application ideas not yet fully revised. On the one hand, the existence of the gap between schema validation in PTIME and $\mathcal{L}^{\text{cons}}$ is not clear yet. Meaning that it is not known if there exists schemas for which validation is tractable that are not consistent. We at least hope that the definition of consistent SHACL contributes to figure out the fully tractable SHACL fragment.

On the other hand, a syntactical translation for SHACL constrains by taking consistency into account is still missing. For example, when proving the hierarchy of SHACL fragments we indirectly showed that the use of disjunction ($\vee$) directly translates as a consistent feature for SHACL. The same can be considered for other traversal operators, similar to $\geq_n p.s$. For example, a "for all neighbors" operator ($\forall p.s$) could be added, that matches all p successors of a node, instead of a certain amount. We believe this too can be showed to produce consistent schemas. On the other hand, the use of the mentioned "exactly n successors" operator ($=_n r.s$) can be shown to produce inconsistent schemas in

some contexts, so its use may be limited, while adding both $\vee$ and $\forall$ as base logical features makes sense. They are direct features that plain SHACL offers, but the implications on the known hierarchy and introduced approaches has to be studied.

REFERENCES

Arenas, M., Gutiérrez, C., & Pérez, J. (2009). Foundations of RDF Databases. In *Reasoning Web. Semantic Technologies for Information Systems* (pp. 158–204).

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The Semantic Web. *Scientific American*, *284*(5), 34–43.

Corman, J., Florenzano, F., Reutter, J. L., & Savkovic, O. (2019). Validating SHACL constraints over a SPARQL endpoint. *ISWC*.

Corman, J., Reutter, J. L., & Savkovic, O. (2018a). Semantics and validation of recursive SHACL. *ISWC*.

Corman, J., Reutter, J. L., & Savkovic, O. (2018b). Semantics and validation of recursive shacl (extended version). *Technical Report KRDB18-1, Free Univ. Bozen-Bolzano*. (`https://www.inf.unibz.it/krdb/tech-reports/`)

Corman, J., Reutter, J. L., & Savkovic, O. (2018c). A tractable notion of stratification for SHACL. In *Iswc*.

Ekaputra, F. J., & Lin, X. (2016). SHACL4p: SHACL constraints validation within Protégé ontology editor. In *ICoDSE*.

Pérez, J., Arenas, M., & Gutiérrez, C. (2009). Semantics and complexity of sparql. *ACM Transactions and Database Systems*, *34*(3).

Shaclex. (n.d.). (`github.com/labra/shaclex/`)

Stardog ICV. (n.d.). (`www.stardog.com/blog/data-quality-with-icv/`)

TopBraid Composer. (n.d.). (www.topquadrant.com/products/topbraid-composer/)