MASTER

Towards Efficient Validation of RDF Graphs against Recursive SHACL

Lahaye, C.R.F.

Award date:
2020

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Database Group

Towards Efficient Validation of RDF Graphs against Recursive SHACL

MSc Thesis

Chris Lahaye

Supervisor:
Dr. George Fletcher

External Supervisor:
Dr. Michael Schmidt (Amazon Web Services, Amazon Neptune)

Assessment Committee:
Dr. George Fletcher
Dr. Michael Schmidt
Dr. Alexander Serebrenik
Dr. Nikolay Yakovets

Eindhoven, February 2020

Acknowledgement

My project has been carried out at the Eindhoven University of Technology in the Database Group of the Department of Mathematics and Computer Science. I would like to take this opportunity to thank a few people for their support during my project. My weekly supervisors have been George Fletcher from the Database Group and Michael Schmidt from Amazon Web Services, Amazon Neptune.

I would like to thank George Fletcher for his continued guidance, advice, and especially, for his support. I would like to thank Michael Schmidt for his extensive willingness and commitment in guiding and advising me at any time. I would also like to thank the Amazon Neptune team, and in particular Michael Schmidt, for our successful collaboration and for providing me with the Amazon Web Services resources needed to successfully carry out my project. Lastly, I would like to thank Alexander Serebrenik and Nikolay Yakovets for serving on my assessment committee, as well as Julien Corman for our fruitful discussions and effort taken clarifying his own research.

Thank you.

Keep exploring

Chris Lahaye

Abstract

RDF [29] is the W3C standard for representing directed, labeled graph data. We research various approaches to validate RDF graphs, in particular, the SPARQL query language for RDF [20] and constraint language SHACL [23]. The problem of SHACL is that its semantics has been defined informally using textual definitions and SPARQL queries, and that recursion, its most distinguishing feature, remains explicitly undefined. We propose practical solutions to efficiently validate SHACL, using the formal semantics for recursive SHACL proposed in [11].

First, we study the theoretical aspects to validating SHACL. It has been shown in previous research that validation of the core of SHACL is intractable. Intractability stems from the ability to validate constraints using arbitrary negation and recursion. This property still holds for constraints with stratified negation and just basic operators. We identify a new tractable recursive fragment with strictly stratified negation and additional native operators for universal quantification and disjunction. It is more expressive than the previously identified tractable recursive fragments [10, 11], allowing for additional constraints to be expressed using strictly stratified negation and native operators to express universal quantification and disjunction without the use of negation.

Validation of recursive schemas is based on a minimal fixed-point assignment. The formal semantics is based on partial assignments, leaving the possibility to assign neither shape nor its negation to nodes. We provide a complete proof of tractability for the fragments that have previously been characterized as tractable, and show that this property still holds for our newly defined fragment. We show that if the minimal fixed-point assignment, which can be computed in polynomial time, does not assign the negated shape to a node targeted by this shape, that there must exist another constraint satisfying assignment that successfully assigns the shape to this node. This implies that a node is valid against a shape if and only if the minimal fixed-point assignment does not assign the negated shape.

Moving on to the practical side, we study the validation complexity of non-referencing constraints, constraints whose validation does not depend on shapes, using a native implementation and by means of a SPARQL query. We identify a few constraint types that may benefit from a native implementation as their analysis showed fewer index lookups or scanned triples compared to their related SPARQL query plans.

We propose a new native algorithm to validate non-recursive SHACL. The validation against referencing constraints is based on the nested validation of nodes against referenced shapes. This recursion may be infinite when the referenced shape is recursive and the node part of a cycle. Connecting previous pieces, we propose a new native hybrid algorithm to validate recursive SHACL and mitigate this problem by extending the algorithm for non-recursive SHACL with a minimal fixed-point algorithm handling validation against recursive shapes. It is sound and complete for all tractable fragments identified so far. Both algorithms run in polynomial time, use the concrete SHACL language, and generate SHACL-like validation reports.

In order to understand how our approach performs in practice, we perform an experimental study of the performance and scalability of both algorithms. Our experiments demonstrate that the validation of large real-world data sets against complex SHACL schemas can be performed efficiently, in the order of seconds. Even selectively adding recursion did not have a significant impact on validation time; our experiments only showed an increase of 3%. This demonstrates the effectiveness of our pruning strategies and minimal overhead by running punctual fixed-point iterations.

Contents

Chapter 1

Introduction

1.1 Motivation

Graph databases are typically used when the interconnectivity or topology of data is of importance [1]. Data is presented in the form of the graph data model where: data is represented by graphs or by data structures generalizing the notion of graphs; data manipulations are expressed by graph transformations or by operations whose main primitives are on graph features [5]. Examples are social networks [34, 8], information networks representing information flows [26, 3], biological networks [4, 7, 16], and Knowledge Graphs [31]. Resource Description Framework (RDF)[29] is such a graph data model that is standardized, widely used, and supported by a large community.

Graph data is typically semi-structured in nature, which is often explained as schemaless or self-describing [2]. There is no separate description of the structure of data. Instead, data is structured following an implicit and often partial schema. Such schemas are not strictly defined or enforced as is the case with relational data. This flexibility leads to higher levels of heterogeneity which may cause issues in conceptually understanding what a data set represents or contains, how its structured, and how it evolves. This complicates working with such data as no schema is defined or enforced, while the structure may mutate over time.

A high-level description capturing the structural properties of the graph and the types and distributions of data facilitates communication, understanding, analysis, data integration, and allows for appropriate integrity constraints to be expressed over the graph structure. It can be used to optimize query evaluation, improve data partitioning and replication, construct indices, and most importantly, to enforce data integrity with the use of constraints [2]. The need for structure increases as graph data is becoming more popular and constantly growing in terms of volume and variety.

1.2 Problem Description

Various constraint languages and validation approaches exist to express constraints and validate the adherence of RDF graphs against these constraints [20, 22, 14, 35, 33, 23]. These languages typically differ in the purpose they were designed for, expressivity of constraints, and complexity of validation.

SPARQL Protocol and RDF Query Language (SPARQL)[20] is a standardized and widely adopted query language for data represented as RDF. This makes SPARQL and the SPARQL-based constraint language SPARQL Inferencing Notation (SPIN)[22] interesting choices for constraint validation. However, using a query-based approach for validation limits the expressivity and performance to that of the query endpoint. The exact performance implications of using query-based validation instead of a native implementation to validate constraints are currently unclear.

The constraint languages Shape Expressions (ShEx)[33] and Shapes Constraint Language (SHACL)[23] have been explicitly designed for constraint checking. ShEx development started under the W3C Data Shapes Working Group with the goal of forming a W3C Recommendation for describing structural constraints and validating RDF instance data against those. The group became divided between the two views, validation as schema recognition and as constraint checking, the ShEx and SHACL group, respectively. The ShEx group split to form the ShEx Community Group [18]. The W3C Data Shapes Working Group later produced SHACL, a language for defining structural constraints on RDF graphs. It successfully became a W3C Recommendation in 2017. SHACL can be seen as the most prominent language for constraint validation due to its standardization, support for recursion, and ability to validate and declare high-level reusable components of arbitrary SPARQL-based constraints.

The problem of SHACL is that its semantics has been defined informally using textual definitions and SPARQL queries, and that recursion, its most distinguishing feature, remains explicitly undefined. The study of [11] proposes formal semantics for recursive SHACL and shows that validation is already intractable for a severely limited fragment. Features such as recursion and arbitrary negation make efficiently validating SHACL a non-trivial task. It is currently unclear how to efficiently validate RDF graphs against a rich fragment of SHACL, while pragmatic algorithms remain unknown.

1.3 Research Question

In our study, we aim to provide an answer to the following research question:

> **How can RDF graphs be efficiently validated against a rich fragment of SHACL that includes recursion?**

Therefore, we investigate the following sub-questions:

- What are the differences between SPARQL-based validation and validation by means of a native implementation in terms of performance?
- What is the complexity of validation of SHACL and how is it affected by specific features?
- How to efficiently validate (recursive) SHACL?

1.4 Contributions

Our contributions are:

- We review existing validation approaches and constraint languages for RDF graphs.
- We propose a new tractable and more expressive recursive SHACL fragment, called strictly stratified $\mathcal{L}^+$. It requires strictly stratified negation, supports all SHACL Core operators, and has additional native operators for universal quantification and disjunction.
- We prove that validation of all tractable recursive SHACL fragments identified so far, in particular strictly stratified $\mathcal{L}^+$, is indeed tractable.
- We study the differences between SPARQL-based validation and a native implementation by studying generated query plans for available SPARQL definitions of SHACL constraints and their potential native implementations. We assess validation performance by reasoning about the number of index lookups and scanned triples, and by doing so, identify a few constraint types that may benefit from a native implementation.
- We propose a new native algorithm for validating non-recursive SHACL. It runs in polynomial time, uses the concrete SHACL language, and generates SHACL-compliant validation reports.

- We propose a new native hybrid algorithm for validating recursive SHACL by extending the algorithm for non-recursive SHACL with a minimal fixed-point algorithm. It is sound and complete for all tractable fragments identified so far, runs in polynomial time, uses the concrete SHACL language, and generates SHACL-like validation reports.
- We provide an implementation of the new native hybrid algorithm in TopBraid's SHACL API. The source code has been made publicly available at `https://github.com/ChrisLahaye/shacl` to contribute to the open-source and academic community.
- We perform an experimental study of the performance and scalability of both algorithms, demonstrating that the validation of large real-world data sets against complex SHACL schemas can be performed efficiently, in the order of seconds, as well as the marginal cost of handling recursion, effectiveness of pruning strategies, and minimal overhead by running punctual fixed-point iterations.

1.5 Organization

Chapter 2 summarizes preliminary knowledge for the remainder of this thesis, in particular, on RDF and SPARQL. Chapter 3 provides a brief overview of existing validation approaches and constraint languages for RDF. In Chapter 4 we focus on the constraint language SHACL. We first introduce the basic concepts of SHACL, provide an overview of constraints, and discuss the abstract syntax, semantics, complexity, and tractable fragments. In Chapter 5 we study the differences between SPARQL-based validation and a native implementation to validate constraints. Chapter 6 describes practical solutions to validating SHACL. We propose algorithms to validate non-recursive and recursive SHACL. In Chapter 7 we discuss the experimental study of the performance and scalability of both algorithms. We also compare our results with another study that takes a different approach to validating SHACL. Lastly, Chapter 8 presents concluding remarks and future work.

Chapter 2

Preliminaries

2.1 Resource Description Framework

Introduction Resource Description Framework (RDF)[29] is a metadata model for representing data by defining the relationships between things. These things are called resources and can be anything, for instance, physical objects, abstract concepts, or values. Resources are denoted by an Internationalized Resource Identifier (IRI $\supset$ URI $\supset$ URL). Values such as strings and numbers are denoted by so-called literals.

RDF defines an abstract syntax that is used to link with all RDF-based languages and specifications such as concrete RDF syntaxes or query languages. Its core data structure is the RDF graph which is a set of triples, each consisting of a subject, predicate, and object. Each triple expresses that some relationship, indicated by the predicate, holds between the resources denoted by the subject and object. Relationships can be expressed about resources without explicitly naming them by using so-called blank nodes as subject or object. Blank nodes can have local identifiers in concrete RDF syntaxes but are limited in scope to a particular graph.

Formally, an RDF triple is a triple of the form (subject, predicate, object) such that:

- subject is an IRI or blank node;
- predicate (also known as property) is an IRI;
- object is an IRI, literal, or blank node.

An RDF graph can be visualized as a node and directed-arc diagram (see Example 1) by drawing the subjects and objects of triples as nodes, and then connecting them by drawing an arc for each triple directed from the subject to the object node with the predicate as label. Nodes for resources are drawn as circles and literals as rectangles.

Namespaces An RDF vocabulary is a collection of IRIs used to describe things. The IRIs in a vocabulary usually start with a common substring called the namespace IRI. Namespace IRIs are often abbreviated by convention into a shorter string, called the namespace prefix. Within this thesis, the following namespaces are used:

Namespace prefix	Namespace IRI
ex:	`http://example.com/ns#`
rdf:	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
rdfs:	`http://www.w3.org/2000/01/rdf-schema#`
spin:	`http://spinrdf.org/spin#`
sp:	`https://spinrdf.org/sp#`
sh:	`http://www.w3.org/ns/shacl#`
dbo:	`http://dbpedia.org/ontology/`

Example 1. The data that James likes Mary and that both have British nationality can be modeled as an RDF graph with the triples:

- (*ex:James*, *ex:likes*, *ex:Mary*)
- (*ex:James*, *ex:nationality*, "British")
- (*ex:Mary*, *ex:nationality*, "British")

where *ex:James* and *ex:Mary* are IRIs denoting James and Mary, respectively, *ex:likes* and *ex:nationality* are IRIs denoting relationships, and "British" is a literal denoting the value British. Figure 2.1 visualizes this RDF graph as a node and directed-arc diagram.

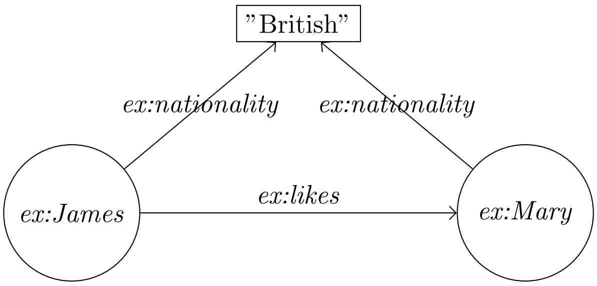


Figure 2.1: A basic RDF graph visualized as a node and directed-arc diagram.

△

Terminology The nodes of an RDF graph are the subjects and objects of its triples. RDF terms are any IRI, literal, or blank node.

Definition 2.1.1. (Property values) The values of a property p for a node v are the objects of triples with v as subject and p as predicate.

Example 2. The nodes of the RDF graph of Example 1 are: *ex:James*, *ex:Mary*, and "British". The value of the property *ex:nationality* for node *ex:James* and *ex:Mary* is "British". △

RDF Vocabulary Description Schema (RDFS)[14] in an extension of RDF and provides mechanisms to describe groups of related resources and the relationships between them. Resources can be assigned to one or more groups, called classes. Resources can declare to be a member of a class using the predicate *rdf:type*. RDFS comes with a set of inference rules allowing to derive new facts from an RDF graph. For instance, if a class is a subclass of another class, then every member of this class is also a member of the other class. Classes can declare to be a subclass of another class using the predicate *rdfs:subClassOf*.

Definition 2.1.2. (Class instance) A node v is an instance of class C if and only if there exists a triple $(v, \textit{rdf:type}, C')$ such that $C' = C$ or C' is a subclass of C, i.e., there exists a sequence of triples $(C_1, \textit{rdfs:subClassOf}, C_2), (C_2, \textit{rdfs:subClassOf}, C_3), ...$ such that the subject of the first triple is C', all subsequent triples are connected by using the object of the previous triple as subject, and the object of the last triple is C.

Example 3. We extend the RDF graph of Example 1 with the data that James is a male and Mary a female by declaring the corresponding resources a member of class *ex:Male* and *ex:Female*, respectively. We add the triples:

- (*ex:James*, *rdf:type*, *ex:Male*)
- (*ex:Mary*, *rdf:type*, *ex:Female*)

We model that all males and females are human by declaring class *ex:Male* and *ex:Female* a subclass of class *ex:Human*. We add the triples:

- (*ex:Male*, *rdfs:subClassOf*, *ex:Human*)
- (*ex:Female*, *rdfs:subClassOf*, *ex:Human*)

The node *ex:James* is now an instance of class *ex:Male* and *ex:Human*, and node *ex:Mary* an instance of class *ex:Female* and *ex:Human*. Figure 2.2 visualizes this RDF graph.

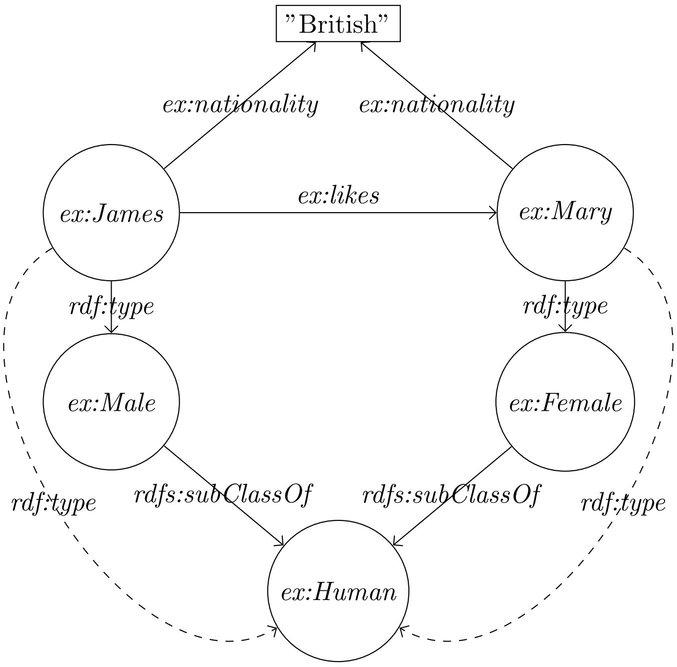


Figure 2.2: An RDF graph using the RDFS vocabulary. Dashed edges represent triples implied by the RDFS inference rules.

△

An RDF collection is a group of things represented as a list structure. The list structure consists of the properties *rdf:first* and *rdf:rest*. The property *rdf:first* denotes the element and *rdf:rest* the remainder of the list. The resource *rdf:nil* represents an empty list and can be used as value for property *rdf:rest* to denote the end of the list.

Example 4. An RDF collection with elements *ex:James* and *ex:Mary* is denoted by the triples:

- ($\_:b_1$, *rdf:first*, *ex:James*)
- ($\_:b_1$, *rdf:rest*, $\_:b_2$)
- ($\_:b_2$, *rdf:first*, *ex:Mary*)
- ($\_:b_2$, *rdf:rest*, *rdf:nil*)

where $\_:b_1$, $\_:b_2$ are two distinguished blank nodes. △

Definition 2.1.3. (List element) A node v is an element of list l, denoted by $v \in l$, if and only if there exists a triple (l, *rdf:first*, v) or a sequence of triples (l_1, *rdf:rest*, l_2), (l_2, *rdf:rest*, l_3), ... such that the subject of the first triple is l, all subsequent triples are connected by using the object of the previous triple as subject, and the object of the last triple has v as value for property *rdf:first*.

Notation In the remainder of this thesis, we write RDF using the concrete RDF syntax Turtle[15] in which triples are written by writing their elements, separated by a white space and terminated by a dot symbol. The subject of the previous triple can be repeated by terminating the triple with a colon symbol. The keyword *a* as predicate is an alternative for *rdf:type*. We omit the mapping from namespace prefix to IRI for readability.

Example 5. The extended RDF graph of Example 3 can be written as follows:

```
ex:James
  a ex:Male ;
  ex:likes ex:Mary ;
  ex:nationality "British" .

ex:Male rdfs:subClassOf ex:Human .

ex:Mary
  a ex:Female ;
  ex:nationality "British" .

ex:Female rdfs:subClassOf ex:Human .
```

△

Blank nodes are written as _:x where x is a label. A blank node with multiple properties can be written using the [p_1 o_1 ; p_i o_i ; p_n o_n] syntax such that p_i is a property and o_i its value. Each property results in a triple $(\_{:}b_1, p_i, o_i)$ where _:b_1 is the blank node. Collections are written using the syntax (a b c) where a, b, and c are its elements.

Example 6. The statement that James likes some female with Dutch nationality is denoted by the triples:

- (*ex:James*, *ex:likes*, _:b_1)
- (_:b_1, *rdf:type*, *ex:Female*)
- (_:b_1, *ex:nationality*, "Dutch")

where _:b_1 is a fresh blank node.

This can be written as follows:

```
ex:James ex:likes [
  a ex:Female ;
  ex:nationality "Dutch"
] .
```

△

2.2 SPARQL Query Language for RDF

Introduction The SPARQL Query Language for RDF [20] is a query language for data represented as RDF. A SPARQL query consists of a set of graph patterns in which each element can be a variable, potentially shared among patterns. Solutions to these variables are then found by matching these graph patterns to RDF triples in the data set. Variables are prefixed with a question mark and can be statically bound using the keyword BIND or VALUES.

SPARQL has four types of queries:

- ASK returns a boolean indicating whether there exists a solution,
- SELECT returns variable bindings of solutions,
- CONSTRUCT returns an RDF graph constructed by substituting the variables in the graph patterns with solutions, and

- DESCRIBE returns an RDF graph providing context (e.g., the neighborhood) for matched resources.

We omit the mapping from namespace prefix to IRI for readability.

Example 7. The question of which females are liked by James can be answered by finding solutions to the x variable in the graph patterns:

- (*ex:James*, *ex:likes*, $?x$)

- ($?x$, *rdf:type*, *ex:Female*)

This translates into the following query:

```
SELECT ?x
WHERE {
  ex:James ex:likes ?x . ?x a ex:Female
}
```

Evaluating this query on the RDF graph of Examples 3 and 5 results in the solutions:

x
ex:Mary

The question of whether such a female exists can be answered by means of the ASK query obtained by replacing the first two lines with `ASK {`. Evaluating this query on the RDF graph results in the boolean value true. △

Combining Graph Patterns The graph patterns A and B can be combined by means of a conjunction `A . B`, joining together the results of A and B by matching the shared variables. An optional graph pattern `A OPTIONAL { B }` joins the results of A with B when possible, but keeps the solutions of A even if there is no match with B. The disjunction `A UNION { B }` includes both the results of A and B. The negation `A MINUS { B }` includes the results of A that do not match with B.

Example 8. We continue with Example 7. When we are also interested in knowing which (first) thing other than potentially James himself is liked by these females (liked by James), but without requiring that such a thing exists, we add an optional graph pattern with negation. This translates into the following query:

```
SELECT ?x ?y
WHERE {
  ex:James ex:likes ?x . ?x a ex:Female
  OPTIONAL { ?x ex:likes ?y MINUS { ?x ex:likes ex:James } }
}
```

Suppose Mary likes James, cycling, and reading, and that we add this data to the RDF graph, then evaluating this query on the extended RDF graph results in the solutions:

x	**y**
ex:Mary	”cycling”

△

Filters The keyword FILTER restricts solutions to those for which an expression evaluates to true. Expressions may include standard mathematical and logical operators. Filters on the (non-)existence of triple patterns can be made using FILTER EXISTS and FILTER NOT EXISTS.

Example 9. Suppose our graph also stores someone's age, and that we are interested in knowing which females are liked by James that are older than 35 and do not like James back. This translates into the following query:

```
SELECT ?x
WHERE {
  ex:James ex:likes ?x . ?x a ex:Female . ?x ex:age ?y
  FILTER(?y > 35) FILTER NOT EXISTS { ?x ex:likes ex:James }
}
```

△

Solutions Sequence Modifiers Solutions sequence modifiers are applied on the unordered collection of solutions generated by query patterns to create another, user desired, collection of solutions. The modifiers that we are using in the remainder of this thesis are: ORDER BY to put the solutions in a specific order, and DISTINCT to ensure solutions are unique.

Aggregates Aggregates apply expressions over groups of solutions. They are used to generate a result computed over a group of solutions, instead of a single solution. The GROUP BY clause is used to group the solutions for which an aggregated value (e.g., COUNT or SUM) is calculated. The clause HAVING filters grouped solution sets.

Example 10. The question of which females, sorted by their nationality, like more than one male can be answered by the following query:

```
SELECT ?x
WHERE {
  ?x a ex:Female . ?x ex:nationality ?y .
  ?x ex:likes ?z . ?z a ex:Male
}
GROUP BY ?x
HAVING(COUNT(?z) > 1)
ORDER BY ?y
```

△

Terminology A property path is a possible route through a graph between two nodes. A trivial case is a property path of length 1. Property paths allow for more concise expression of basic graph patterns.

Example 11. The nationalities of things liked by James can be determined by finding solutions to the o variable in the (basic) graph patterns:

- (*ex:James*, *ex:likes*, $?x$)
- ($?x$, *ex:nationality*, $?o$)

These basic graph patterns can be translated into a sequence path of IRI *ex:likes* followed by *ex:nationality*. This translates into the following query:

```
SELECT ?o
WHERE {
  ex:James ex:likes/ex:nationality ?o
}
```

Evaluating this query on the RDF graph of Examples 1, 3 and 5 results in the solutions:

o
”British”

△

Definition 2.2.1. (Property path values) The values of a property path p for a node v are the solutions to variable o in the result of the query *SELECT ?o WHERE* { ?s p ?o } such that p is substituted and variable s bound or substituted with v.

Example 12. The values of property path *ex:likes/ex:nationality* for node *ex:James* in the RDF graph of Examples 1, 3 and 5, i.e. the solutions to variable o in the result of the query in Example 9, is the set consisting of a single value "British". △

Chapter 3

Validation Approaches and Constraint Languages

3.1 Introduction

Constraint validation approaches for graphs differ in the languages used to express constraints and in how these languages are implemented. Validation approaches using constraint languages can be categorized into rule languages, query-based constraint languages, and grammar-based constraint languages, and their implementations into hard-coded systems, reasoners, and query endpoints.

Validation approaches without any constraint language are implemented by a hard-coded system that implements both the description and validation of constraints. They typically perform well, allow for optimization, but lack in customization as constraints cannot be expressed in a declarative manner.

Rule-based approaches define constraints using vocabularies and ontologies which are validated using a reasoner or query endpoint. Examples are RDFS [14] and Ontology Language (OWL)[35].

Query-based approaches allow constraints to be expressed in the form of queries or by means of a high-level language and are validated using a query endpoint. They allow customization by defining additional queries or constructs in the language. Examples are SPARQL [20], SPIN [22], and SHACL [23].

Grammar-based approaches define a domain specific language to declare constraints which are validated using a query endpoint or hard-coded system. They allow customization by defining additional constructs in the language. High-level constraint languages are comparatively easy to understand and constraints can be formulated concisely. Examples are ShEx [33] and SHACL [23].

We briefly discuss some of these approaches in the following sections.

3.2 Rule-based Reasoner

Introduction A rule-based reasoner is a system that receives a set of facts and rules in the form of if-then statements and produces new facts by inference. There are two main strategies for reasoning, forward and backward chaining. Forward chaining starts with the known facts, and infers all possible facts. Backward chaining starts with the consequent and performs backwards to infer the needed antecedent. Implementations may differ in supported logics, built-in functions, strategies, and features (e.g., proof explanation and tracing). Support for expressive built-ins are needed as validation often deals with string comparisons and mathematical calculations [6].

Constraints can be expressed in the form of rules in the supported logic, or implicitly by using vocabularies and ontologies like RDFS [14] and OWL [35]. These vocabularies come with their own interpretation that indicate how a statement should be interpreted and which new facts can be derived. Under the right semantics, this can be used to construct rules that infer constraint violations.

Rule-based reasoners generate logic proofs stating how a new fact representing a constraint violation was inferred. They support adding inferencing steps by adding custom rules, hereby allowing customization. They only need a single system to declare the constraints and the set of inferencing rules.

Semantics Validation and inference typically assume different semantics that may lead to different results depending on the type of constraint. Reasoning requires presence of Open World Assumption (OWA) and absence of Unique Name Assumption (UNA), whereas validation typically depends on Closed World Assumption (CWA) and presence of UNA [9, 18]. As a consequence, efforts have been made into combining both OWA and CWA [19, 30], allowing parts of the world to be explicitly closed. This is called Local Closed World Reasoning. An alternative approach to using reasoners for constraint validation is to support Scoped Negation as Failure (SNAF) instead of CWA and predicates to compare URIs and literals instead of UNA [6]. The Open World Assumption (OWA) is the assumption that the truth value of a statement may be true irrespective whether it is known to be true. This in contrast to a Closed World Assumption (CWA) where failure to infer a statement implies it to be false and its negation to be true. Example 13 demonstrates the difference between validating a constraint with OWA and CWA.

Example 13. Consider the constraint that every human must have a father and assume there exists a human without father. Under OWA this constraint is not violated as there may exist an unknown father for this human. So, under OWA a constraint is only violated if there exists a contradiction. △

The Unique Name Assumption (UNA) is the assumption that different names always refer to different entities, meaning that two different URIs could refer to the same entity. The absence of UNA may lead to non-intuitive inference and validation as shown in Example 14.

Example 14. Consider the constraint that limits the cardinality of a property to one. If a resource has two different values for this property, then this is not a violation as they could refer to the same entity. As result, a reasoner will infer that the two resources are the same while this may not necessarily be the case. △

3.3 SPARQL-based Constraint Validation

SPARQL can be used for constraint validation by expressing constraints as ASK queries. The expressivity of SPARQL is equivalent to that of relation algebra and the complexity of query evaluation is PSPACE [27]. These properties and its wide adoption as standard query language make SPARQL an interesting choice for constraint validation. However, queries can become quite complex and long as SPARQL does not allow any reusable high-level constructs. SPARQL does not support recursive constraints, unless the recursive SPARQL extension introduced in [28] is used.

Example 15. The following query determines whether there exists a violation to the constraint that every human must have a father:

```
ASK {
  ?x a ex:Human
  FILTER NOT EXISTS { ?x ex:father ?y }
}
```

△

Various work exists that propose RDF vocabularies to express constraints and then validate these constraints by mapping them into SPARQL queries [17, 25]. In [17] they propose an RDF vocabulary to express basic RDF constraints and a mapping to validate these constraints using SPARQL. In [25] they map relational data to RDF while preserving the primary and foreign key constraints in a dedicated vocabulary. They provide mappings from these key constraints to SPARQL queries for validation.

3.4 SPARQL Inferencing Notation

SPARQL Inferencing Notation (SPIN)[22] is a SPARQL-based rule and constraint language for RDF graphs. It can be used for various purposes like making automatic calculations, performing constraint validation with closed world semantics, and to isolate rules under specific conditions.

Rules are implemented using SPARQL CONSTRUCT queries or DELETE/INSERT update operations, and constraints using SPARQL ASK/CONSTRUCT queries or SPIN templates. Templates are parameterized and reusable high-level constructs of SPARQL queries.

The properties *spin:rule* and *spin:constraint* are used to link rules and constraints, respectively, to a class such that it is applied to each instance of this class. Each ASK query defines a constraint and is evaluated for each instance of the linked class. A node violates the constraint if the ASK query evaluates to true.

Example 16. The constraint that every human must have a father can be expressed in SPIN as follows:

```
ex:Human spin:constraint [
  a sp:Ask ;
  rdfs:comment "must␣have␣a␣father" ;
  sp:where ([
    a sp:NotExists ;
    sp:elements ([
      sp:subject spin:_this ;
      sp:predicate ex:father ;
      sp:object [ sp:varName "y" ]
    ])
  ])
] .
```

△

The SPIN documentation [24] refers to the Shapes Constraint Language (SHACL)[23] as its legitimate successor, stating that every SPIN feature is supported and improved by the constraint language SHACL. Chapter 4 discusses SHACL in detail.

Chapter 4

Shapes Constraint Language

4.1 Introduction

Shapes Constraint Language (SHACL)[23] is an RDF-based schema language to constrain, validate, and infer RDF graphs. It became a W3C recommendation in 2017 and consists of two parts:

1. SHACL Core describes a core RDF vocabulary to define shapes and constraints; and
2. SHACL-SPARQL extends SHACL Core with SPARQL-based constraints and an extension mechanism to declare new reusable components.

Our focus is primarily on SHACL Core. SHACL Core evolves around three concepts: shapes, focus nodes, and constraints. Simply stated, focus nodes are RDF terms being validated against shapes with respect to the constraints they declare.

A shape is a node-centric collection of constraints. In line with this idea, SHACL Core defined two types of shapes: node and property shapes. Node shapes declare constraints that specify requirements on the focus node itself, whereas property shapes specify requirements on the values of a particular property or path for the focus node.

The nodes that must meet the requirements are called value nodes. The value nodes for constraints declared by a node shape is the set with as only element the focus node, and for a property shape the set of nodes reachable from the focus node with the declared property or path.

The syntax of SHACL is RDF. Shapes and other constructs can be expressed in the form of an RDF graph. These graphs are called shapes graphs and contain zero or more shapes. The RDF graphs that are validated against a shapes graph are called data graphs.

Example 17. The following shapes graph expresses that every human must have a father:

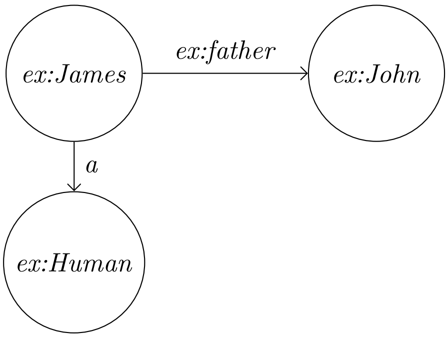
```
ex:HumanShape
  a sh:NodeShape ;
  sh:targetClass ex:Human ;
  sh:property [
    sh:path ex:father ;
    sh:minCount 1
  ] .
```

This shapes graph declares the node shape *ex:HumanShape.* This shape targets instances of class *ex:Human*, requiring each instance to be valid against this shape. Each instance is validated as focus node against the constraints declared by the shape.

The node shape declares a property constraint as it has a value for its mandatory parameter, the property *sh:property.* The property constraint requires that the value nodes for a given focus node (an instance of *ex:Human*) is valid against the property shape given as value for this parameter. The property constraint has been declared by a node shape, therefore, the set of value nodes is the set with as only element the given focus node. Each value node is validated as focus node against the constraints declared by the property shape.

The property shape declares a minimum cardinality constraint as it has a value for its mandatory parameter, the property *sh:minCount*. The minimum cardinality constraint requires that the number of value nodes for the given focus node is greater than the number given as value for this parameter. As the minimum cardinality constraint has been declared by a property shape, the set of value nodes are the nodes reachable from the given focus node over the declared property or path (the value of property *sh:path* for the property shape).

An RDF graph that is valid with respect to the shapes graph is the following:



△

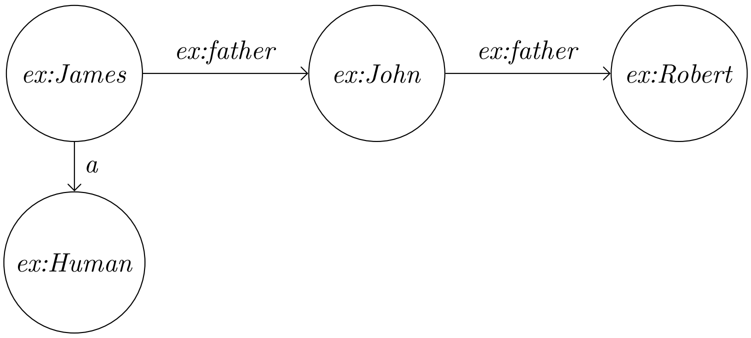
Shapes can declare targets that identify focus nodes. SHACL Core includes the following kinds of targets: node targets that target specific nodes identified by fixed IRIs, class-based targets that target instances of a specific class (see Example 17), implicit class targets that target instances of the shape as class, subjects-of targets that target the subjects of triples with a specific predicate, and objects-of targets that target the objects of triples with a specific predicate. Shapes that declare one or more target are called target-declaring and their target is the union of all focus nodes identified by each target declaration.

Terminology With the values of (property) path p we mean the values of SPARQL property path p' such that SHACL property path p maps to SPARQL property path p' per the rules defined in [23]. With a list and its members we mean an RDF collection and its elements.

Example 18. The following shapes graph expresses that every human must have a grandfather:

```
ex:HumanShape
  a sh:NodeShape ;
  sh:targetClass ex:Human ;
  sh:property [
    sh:path ( ex:father ex:father ) ;
    sh:minCount 1
  ] .
```

We make the graph of Example 17 valid by adding Robert, the grandfather of James, as father of James' father John, i.e. we add the triple (*ex:John*, *ex:father*, *ex:Robert*):



The values of (SHACL) property path *(ex:father ex:father)* for node *ex:James* are the values of the mapped SPARQL sequence path *ex:father/ex:father* for node *ex:James*, i.e. *ex:Robert*. △

Definition 4.1.1. [23](Shape) A shape is an IRI or blank node s that satisfies at least one of the following conditions:

1. s is an instance of class *sh:NodeShape* or *sh:PropertyShape*.
2. s is the subject of a triple with predicate *sh:targetClass*, *sh:targetNode*, *sh:targetObjectsOf*, or *sh:targetSubjectsOf*.
3. s is the subject of a triple with a constraint parameter as predicate (see Section 4.2).
4. s is a value of a shape-expecting non-list-taking parameter, or a member of a shape-expecting list-taking parameter (see Sections 4.2 and 4.3).

Example 19. The shapes graph of Example 18 declares a shape *ex:HumanShape* as this IRI satisfies condition 1, 2, and 3, and another shape denoted by the blank node which satisfies condition 3 and 4. △

Definition 4.1.2. [23](Property shape) A shape s is a property shape if and only if it has a SHACL property path (per the rules defined in [23]) as only value for the property *sh:path*. Note that it is not required, but recommended, for s to be an instance of class *sh:PropertyShape*.

Example 20. The shape *ex:HumanShape* in the shapes graph of Example 18 is a node shape since it has no value for the property *sh:path*. The shape denoted by the blank node is a property shape as it has a SHACL property path as only value for the property *sh:path*. △

4.2 Constraints

Shapes can declare constraints using the parameters of constraint components. Constraints are instances of constraint components and provide values for their mandatory and potentially optional parameters. Each constraint component is identified by an IRI in the SHACL namespace. In the remainder of this thesis, we omit the *sh:* namespace prefix, for instance, we write *ClassConstraintComponent* instead of *sh:ClassConstraintComponent*.

SHACL Core supports the following type of constraints:

- value type constraints which restrict the type of value nodes (e.g., to be an instance of class *ex:Human*);
- cardinality constraints which restrict the number of values nodes (e.g., to be greater than 1);
- value range constraints which specify range conditions on the value nodes (e.g., being greater than the number 18);

- string-based constraints which specify conditions on the string representation of value nodes (e.g., having at most 2 characters);
- property pair constraints which specify conditions on the sets of value nodes in relation to other properties (e.g., being disjoint with the values of property *ex:dislikes*);
- logical constraints implement common logical operators on shapes (e.g., conjunction);
- shape-based constraints specify complex conditions by validating the value nodes against shapes (e.g., being valid against the node shape *ex:HumanShape*);
- closed shapes which require properties of value nodes to be explicitly enumerated; and
- non-validating constraints (e.g., providing human-readable labels).

The validation of both logical and shape-based constraint components is defined with respect to the validity of value nodes against shapes. Let them be called referencing constraint components.

Definition 4.2.1. (Referencing constraint components) The set of referencing constraint components is the union of logical and shape-based constraint components.

Each constraint component belongs to one of the following types:

1. referencing constraint component requiring validation,
2. referencing constraint component requiring conformance checking, or
3. non-referencing constraint component.

Both referencing constraint components requiring validation and conformance checking have exactly one mandatory parameter that receives a shape or list of shapes as value. The validation against these constraint components is defined with respect to the validity of value nodes against these shapes, i.e. on the nested validation. However, the validation results of validating value nodes against referencing constraint components requiring validation are the results of validating these nodes against the referenced shapes, whereas the validation results of validating against referencing constraint components requiring conformance checking are based on whether these nodes conform to the referenced shapes, i.e. whether the nested validation results are empty.

Example 21. The node shape *ex:HumanShape* in the shapes graph of Example 18 declares a constraint of kind *PropertyConstraintComponent* as it has a value for its mandatory parameter, the property *sh:property.* This is a referencing constraint component requiring validation. The property shape denoted by the blank node declares a constraint of kind *MinCountConstraintComponent* as it has a value for its mandatory parameter, the property *sh:minCount.* This is a non-referencing constraint component. △

The following paragraphs describe the constraint components used in the remainder of this thesis, as well as their type, parameters, and validation results. Table 4.1 describes all SHACL Core constraint components with their parameters and type.

DisjointConstraintComponent Restricts the set of value nodes to be disjoint with the set of nodes that have the focus node as subject and the value of the given property (the value of mandatory parameter *sh:disjoint*) as predicate. If a value node exists as a value of the given property, there is a validation result with the value node as value. This is a non-referencing constraint component.

EqualsConstraintComponent Restricts the set of value nodes to equal the set of nodes that have the focus node as subject and the value of the given property (the value of mandatory parameter *sh:equals*) as predicate. If a value node does not exist as a value of the given property, there is a validation result with the value node as value. If a value of the given property is not a value node, there exists a validation result with the value as value. This is a non-referencing constraint component.

HasValueConstraintComponent Restricts that one of the value nodes is the given RDF node (the value of mandatory parameter *sh:hasValue*). If such value node does not exists, there is a validation result. This is a non-referencing constraint component.

LessThanConstraintComponent Restricts every value node to be smaller than all nodes that have the focus node as subject and the value of the given property (the value of mandatory parameter *sh:lessThan*) as predicate. This is a non-referencing constraint component.

LessThanOrEqualsConstraintComponent Restricts every value node to be smaller or equal than all nodes that have the focus node as subject and the value of the given property (the value of mandatory parameter *sh:lessThanOrEquals*) as predicate. This is a non-referencing constraint component.

MaxCountConstraintComponent Restricts the maximum number of value nodes. If the number of value nodes is greater than the given maximum number (the value of mandatory parameter *sh:maxCount*), there is a validation result. This is a non-referencing constraint component.

MinCountConstraintComponent Restricts the minimum number of value nodes. If the number of value nodes is less than the given minimum number (the value of mandatory parameter *sh:minCount*), there is a validation result. This is a non-referencing constraint component.

NodeConstraintComponent Restricts the set of value nodes to conform to the given node shape (the value of mandatory parameter *sh:node*). If a value node does not conform to the given node shape, there is a validation result with the value node as value. This is a referencing constraint component requiring conformance checking.

PropertyConstraintComponent Restricts the set of value nodes to have the given property shape (the value of mandatory parameter *sh:property*). The validation results are the results of validating the set of value nodes as focus nodes against the given property shape. This is a referencing constraint component requiring validation.

QualifiedMinCountConstraintComponent Restricts the minimum number of value nodes that conform to the given shape (the value of mandatory parameter *sh:qualifiedValueShape*). If the number of value nodes that conform to the given shape is less than the given minimum number (the value of mandatory parameter *sh:qualifiedMinCount*), there is a validation result. This is a referencing constraint component requiring conformance checking.

Table 4.1: List of all SHACL Core constraint components with their parameters and type

The first column lists a short name to uniquely identify the constraint component. This name corresponds to the IRI of the constraint component in the SHACL namespace (e.g., MinCount corresponds to *sh:MinCountConstraintComponent*). The second column lists a description of the constraint component and its parameters. All constraint parameters are mandatory unless explicitly stated otherwise.

Component	**Descriptions and Constraint Parameters**
Non-referencing constraint components	
Class	Restricts each value node to be an instance of the given class.
	sh:class - The class IRI

Closed	Restricts each value node's properties to be explicitly enumerated by one of the property shapes declared as value of property *sh:property*.
	sh:closed - Boolean indicating whether to close the shape
	sh:ignoredProperties - Optional list of IRIs to permit besides those explicitly enumerated
DataType	Restricts each value node to be of the given data type.
	sh:datatype - The data type IRI
Disjoint	Restricts set of value nodes to be disjoint with the set of values of the given property.
	sh:disjoint - The property IRI
Equals	Restricts the set of value nodes to equal the set of values of the given property.
	sh:equals - The property IRI
HasValue	Restricts the given value node to exist.
	sh:hasValue - The RDF term
In	Restricts each value node to be explicitly given.
	sh:hasValue - List of RDF terms
LanguageIn	Restricts each value node's language tag to be explicitly given.
	sh:languageIn - List of strings
LessThan	Restricts each value node to be arithmetically strictly smaller than all values of the given property.
	sh:lessThan - The property IRI
LessThanOrEquals	Restricts each value node to be arithmetically smaller than all values of the given property.
	sh:lessThanOrEquals - The property IRI
MaxCount	Restricts the maximum number of value nodes to the given number.
	sh:maxCount - The maximum cardinality
MaxExclusive	Restricts each value node to be arithmetically strictly greater than the given number.
	sh:maxExclusive - The number
MaxInclusive	Restricts each value node to be arithmetically non-strictly greater than the given number.
	sh:maxInclusive - The number
MaxLength	Restricts the maximum string length of each value node to the given length.
	sh:maxLength - The maximum string length
MinCount	Restricts the minimum number of value nodes to the given number.
	sh:minCount - The minimum cardinality
MinExclusive	Restricts each value node to be arithmetically strictly smaller than the given number.
	sh:minExclusive - The number
MinInclusive	Restricts each value node to be arithmetically non-strictly smaller than the given number.
	sh:minInclusive - The number
MinLength	Restricts the minimum string length of each value node to the given length.
	sh:minLength - The minimum string length
NodeKind	Restricts each value node to be of the given node kind.
	sh:nodeKind - The node kind IRI

Pattern	Restricts each value node to match the given regular expression.
	sh:pattern - The regular expression string
	sh:flags - Optional flags string
UniqueLang	Restricts each value node to have a unique language tag.
	sh:uniqueLang - Boolean indicating whether to require unique language tags
SPARQL	Restricts the solutions in the result of executing the given SELECT query for each value node to have the variable failure bound to true.
	sh:sparql - IRI or blank node with a single value for property *sh:select*
Referencing constraint components requiring conformance checking	
And	Restricts each value node to conform to all given shapes.
	sh:and - List of shapes
Node	Restricts each value node to conform to the given node shape.
	sh:node - The node shape
Not	Restricts each value node to not conform to the given shape.
	sh:not - The shape
Or	Restricts each value node to conform to at least one of the given shapes.
	sh:or - List of shapes
Xone	Restricts each value node to conform to exactly one of the given shapes.
	sh:xone - List of shapes
QualifiedMaxCount	Restricts the maximum number of value nodes that conform to the given shape to the given number.
	sh:qualifiedMaxCount - The maximum number
	sh:qualifiedValueShape - The shape
	sh:qualifiedValueShapesDisjoint - Optional boolean indicating whether only value nodes that do not conform to sibling shapes[1] are counted
QualifiedMinCount	Restricts the minimum number of value nodes that conform to the given shape to the given number.
	sh:qualifiedMinCount - The minimum number
	sh:qualifiedValueShape - The shape
	sh:qualifiedValueShapesDisjoint - Optional boolean indicating whether only value nodes that do not conform to sibling shapes[2] are counted
Referencing constraint components requiring validation	
Property	Restricts each value node to have the given property shape.
	sh:property - The property shape

[1] In the context of *QualifiedMaxCountConstraintComponent* and *QualifiedMinCountConstraintComponent*, the set of sibling shapes is the set of all values of the SPARQL property path *sh:property/sh:qualifiedValueShape* for any shape with the constraint-declaring shape as value of property *sh:property*, minus the value of property *sh:qualifiedValueShape* for the constraint-declaring shape itself.

[2] See footnote 1

4.3 Shape References

Shapes can reference other shapes such that a node is validated against that shape without being targeted directly by that specific shape. A shape's references are the values of shape-expecting non-list-taking constraint parameters (i.e. *sh:not*, *sh:property*, *sh:qualifiedValueShape*, and *sh:node*) and members of lists that are values of shape-expecting list-taking constraint parameters (i.e. *sh:and*, *sh:or*, and *sh:xone*). These are parameters of the referencing constraint components. Recursion refers to a reference cycle such that a shape references itself, directly or via other shapes.

If a shape s_1 references a shape s_2, then we write $s_1 \rightarrow s_2$. If this reference is negated (i.e. a value of constraint parameter *sh:not*), then we write $s_1 \xrightarrow{-} s_2$, else $s_1 \xrightarrow{+} s_2$. With $s_1 \twoheadrightarrow s_2$ we denote that s_2 is referenced by s_1, directly or via other shapes. Similarly for $s_1 \overset{-}{\twoheadrightarrow} s_2$ and $s_1 \overset{+}{\twoheadrightarrow} s_2$, but then with only negated and non-negated references, respectively.

Definition 4.3.1. (References) The set of referenced node shapes $\mathcal{R}(G, s)$ of a shape s in shapes graph G is the union of the values of shape-expecting non-list-taking constraint parameters and members of lists that are values of shape-expecting list-taking constraint parameters. Let $\mathcal{R}(G, s)$ be defined as:

$$\begin{aligned}\mathcal{R}(G,s) = &\{s \xrightarrow{+} s' \mid (s,p,l) \in G \land p \in \{\text{sh:and}, \text{sh:or}, \text{sh:xone}\} \land s' \in l\}\\ &\cup \{s \xrightarrow{+} s' \mid (s,p,s') \in G \land p \in \{\text{sh:qualifiedValueShape}, \text{sh:node}, \text{sh:property}\}\}\\ &\cup \{s \xrightarrow{-} s' \mid (s, \text{sh:not}, s') \in G\}\end{aligned}$$

such that $s' \in l$ if and only if l is an RDF collection of which s' is an element.

Example 22. The following shapes graph G expresses the requirements:

1. Every human must (line 4) have a father (line 10).
2. Every human must not (line 5) have been bitten by a vampire (lines 15 and 16).
3. A vampire has (line 20) been bitten by a vampire (lines 15 and 16) and has (line 21) a father (line 10).

```
1  ex:HumanShape
2    a sh:NodeShape ;
3    sh:targetClass ex:Human ;
4    sh:property ex:FatherShape ;
5    sh:not ex:BittenShape .
6
7  ex:FatherShape
8    a sh:PropertyShape ;
9    sh:path ex:father ;
10   sh:minCount 1 .
11
12 ex:BittenShape
13   a sh:PropertyShape ;
14   sh:path ex:bitten ;
15   sh:qualifiedValueShape ex:VampireShape ;
16   sh:qualifiedMinCount 1 .
17
18 ex:VampireShape
19   a sh:NodeShape ;
20   sh:property ex:BittenShape ;
21   sh:property ex:FatherShape .
```

The references of shapes in this shapes graph are:

$$\begin{aligned}
\mathcal{R}(G, \textit{ex:HumanShape}) &= \left\{ \begin{aligned} &\textit{ex:HumanShape} \xrightarrow{+} \textit{ex:FatherShape}, \\ &\textit{ex:HumanShape} \xrightarrow{-} \textit{ex:BittenShape} \end{aligned} \right\} \\
\mathcal{R}(G, \textit{ex:FatherShape}) &= \emptyset \\
\mathcal{R}(G, \textit{ex:BittenShape}) &= \{\textit{ex:BittenShape} \xrightarrow{+} \textit{ex:VampireShape}\} \\
\mathcal{R}(G, \textit{ex:VampireShape}) &= \left\{ \begin{aligned} &\textit{ex:VampireShape} \xrightarrow{+} \textit{ex:BittenShape}, \\ &\textit{ex:VampireShape} \xrightarrow{+} \textit{ex:FatherShape} \end{aligned} \right\}
\end{aligned}$$

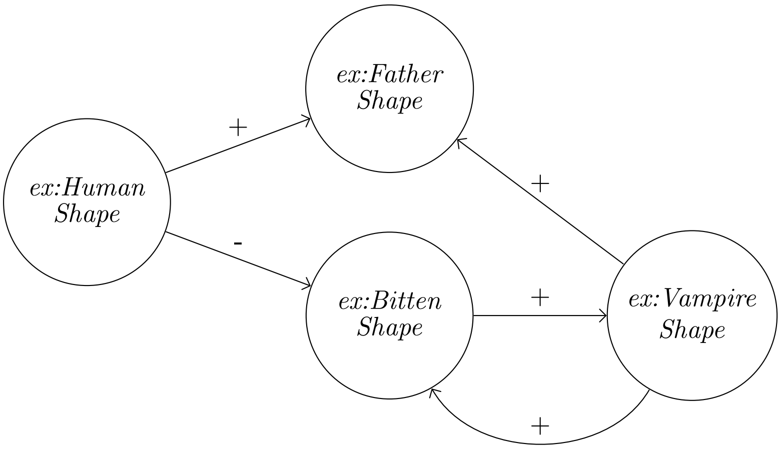
△

Definition 4.3.2. (Dependency graph) The dependency graph P_S of a set of shapes S is the labeled graph whose nodes are the shapes in S with a directed edge for each shape reference labeled with its parity. For each $s_1, s_2 \in S$:

- If $s_1 \xrightarrow{+} s_2$, then there is a directed edge from s_1 to s_2 with label $+$ (called a positive edge).
- If $s_1 \xrightarrow{-} s_2$, then there is a directed edge from s_1 to s_2 with label $-$ (called a negative edge).

The dependency graph can be constructed in $O(|V| + |E|)$ time.

Example 23. The dependency graph of the shapes in shapes graph G of Example 22 is the following:

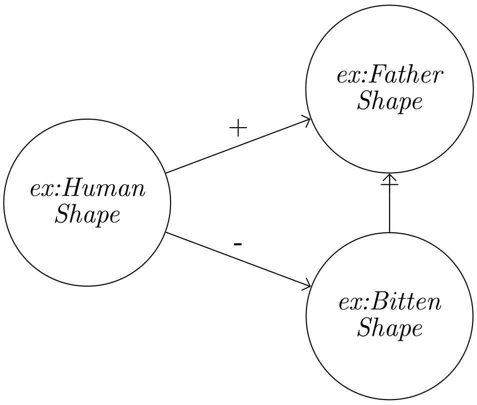


△

Definition 4.3.3. (Positive contracted dependency graph) The positive contracted dependency graph P_S^{SSC+} of a set of shapes S is the dependency graph P_S with each strongly connected component consisting of only positives edges contracted into a single node.

Several algorithms (e.g., Kosaraju's algorithm and Tarjan's strongly connected components algorithm) based on Depth First Search (DFS) compute strongly connected components in $O(|V|+|E|)$ time.

Example 24. The subgraph of the dependency graph of Example 23 consisting of node *ex:BittenShape* and *ex:VampireShape* (and the edges between those nodes) is the only non-trivial strongly connected component consisting of only positive edges. We retract this subgraph into a single node. The positive contracted dependency graph of the shapes in shapes graph G of Example 22 is the following:



△

4.4 Abstract Syntax and Semantics

4.4.1 Introduction

The SHACL syntax and semantics have been informally defined in [23]. The specification consists of SPARQL queries and textual definitions, even though SPARQL is not required for the implementation of SHACL Core. The semantics of validation with recursive shapes is left explicitly undefined.

Corman, Reutter and Savković describe a formal semantics in [11] for the core constraint components of SHACL including recursion. Validation of recursive shapes is based on assignments where nodes are assigned positive or negated shape labels indicating whether nodes conforms to those shapes or not.

They illustrate that when using non-stratified constraints, constraints with recursive references in the scope of negation, a total assignment where every shape must be assigned positively or negatively may render certain graphs invalid. This is the case for any set of shapes containing a reference cycle, and such that an odd number of references in this cycle are in the scope of a negation [11]. Therefore, they propose semantics based on partial assignments such that its possible to neither assign a shape nor its negation. The proposed semantics support arbitrary recursion and negation, can handle simultaneous validation of multiple targets, and is compliant with the standard in the non-recursive case.

Definition 4.4.1. (Stratification) A set of shapes S is stratified if there is a total function $\alpha : S \to \mathbb{N}$ such that:

i If $s_2 \xrightarrow{+} s_1$, then $\alpha(s_1) \leq \alpha(s_2)$.

ii If $s_2 \xrightarrow{-} s_1$, then $\alpha(s_1) < \alpha(s_2)$.

Example 25. The shapes in shapes graph G of Example 22 are stratified as the mapping:

$$\alpha = \{\textit{ex:BittenShape} \rightarrow 0, \textit{ex:VampireShape} \rightarrow 0, \textit{ex:FatherShape} \rightarrow 0, \textit{ex:HumanShape} \rightarrow 1\}$$

satisfies condition (i) and (ii), i.e. is a solution to the system:

$$\begin{cases} \alpha(\textit{ex:FatherShape}) \leq \alpha(\textit{ex:HumanShape}) \\ \alpha(\textit{ex:BittenShape}) < \alpha(\textit{ex:HumanShape}) \\ \alpha(\textit{ex:BittenShape}) \leq \alpha(\textit{ex:VampireShape}) \\ \alpha(\textit{ex:VampireShape}) \leq \alpha(\textit{ex:BittenShape}) \\ \alpha(\textit{ex:VampireShape}) \leq \alpha(\textit{ex:FatherShape}) \end{cases}$$

△

Lemma 1. *Let S be a stratified set of shapes with mapping α and P_S as its dependency graph. If there exists a path from s_1 to s_2 in P_S of only positive edges, then $\alpha(s_2) \leq \alpha(s_1)$; and if there is a path from s_1 to s_2 in P_S containing some negative edge, then $\alpha(s_2) < \alpha(s_1)$.*

Proof. If there exists a path from s_1 to s_2 in P_S of only positive edges, then by Definition 4.3.2, $s_1 \overset{+}{\twoheadrightarrow} s_2$, then $\alpha(s_2) \leq \alpha(s_1)$ by Definition 4.4.1 and transitivity. Similarly for a path with a negative edge. □

The dependency graph can be used to determine if a set of shapes is stratified.

Proposition 4.4.1. *A set of shapes S is stratified if and only if its dependency graph has no cycle containing a negative edge.*

Proof. Suppose the set of shapes S is stratified. In order to derive a contradiction, suppose its dependency graph has a cycle $s_1, ..., s_m, s_1$ with a negative edge from s_m to s_1. Then by Lemma 1, $\alpha(s_1) < \alpha(s_1)$. This is a contradiction, so a cycle containing a negative edge cannot exist if S is stratified. Suppose the dependency graph has no cycle containing a negative edge. In order to derive a contradiction, suppose the set of shapes S is not stratified. Then by Definition 4.4.1, $s_1 \twoheadrightarrow s_m \overset{-}{\rightarrow} s_1$. Then by Definition 4.3.2, the dependency graph has a cycle $s_1, ..., s_m, s_1$ with a negative edge from s_m to s_1. This is a contradiction, so S must be stratified if the dependency graph has no cycle containing a negative edge. □

DFS can be used to detect a cycle in the dependency graph. Thus, determining if a set of shapes S is stratified can be performed in $O(|V| + |E|)$ time.

4.4.2 Abstract Syntax

We work with a logical abstraction of SHACL Core as defined in [11]. It uses a fragment of first-order logic to define constraints. This fragment is called $\boldsymbol{\mathcal{L}}$ and is defined by the grammar:

$\phi ::= \top \mid s \mid I \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \geq_n r.\phi \mid \mathrm{EQ}(r_1, r_2)$

where s is a shape name; I is an IRI; r, r_1, and r_2 are property paths; and $n \in \mathbb{N}^+$. The fragment abstracts away from constraints on IRIs and literals (e.g., regular expression, datatype, value comparison).

Informally: $\top$ always validates; s validates if the shape s is assigned to the target node; $\phi_1 \wedge \phi_2$ validates if both ϕ_1 and ϕ_2 validate; I validates if the target node's IRI matches I; $\neg\phi$ validates when ϕ does not; $\geq_n r.\phi$ validates when more than n nodes, each reachable via r, validate ϕ; $\mathrm{EQ}(r_1, r_2)$ validates when the set of subject-object pairs for property r_1 and r_2 are identical.

The abstract syntax does not capture the target of shapes, which instead, will be specified orthogonally. We refer to [11] for the mapping between SHACL and the abstract syntax.

Example 26. The shapes graph of Example 22 written in the abstract syntax is:

$$\begin{aligned}\phi_{\text{HumanShape}} &= \geq_1 \textit{ex:father}.\top \wedge \neg(\geq_1 \textit{ex:bitten}.\text{VampireShape})\\ \phi_{\text{VampireShape}} &= \geq_1 \textit{ex:father}.\top \wedge \geq_1 \textit{ex:bitten}.\text{VampireShape}\end{aligned}$$

We denote shapes with a name instead of an IRI or blank node, and ϕ_s is the constraint associated with shape s. For each node shape, we introduced a constraint defined as the conjunction of constraints declared by this shape, written in the abstract syntax. △

4.4.3 Semantics

We borrow the notation introduced in [11]. The evaluation $[\![\phi]\!]^{v,G,\sigma}$ of constraint formula ϕ at node v in graph G given assignment σ is defined in Table 4.2. It uses a 3-valued logic in which 0 and 1 represent true and false, respectively, and 0.5 an unknown truth value.

If r is a property path and G a graph, then $r(G)$ denotes the evaluation of r, which consists of all pairs (v, v') of nodes in G such that there is a path from v to v' satisfying r, i.e. $(v, r, v') \in G$. We use $|X|$ to denote the size of structure X.

Definition 4.4.2. [11](Assignment) Let N be the set of shape names. An assignment σ is a total function mapping nodes to subsets of $N \cup \{\neg s \mid s \in N\}$ such that s and $\neg s$ cannot be both in $\sigma(v)$ for any node v.

Table 4.2: Inductive evaluation of constraint formula ϕ at node v in graph G given assignment σ [11]

$$\begin{aligned}
[\![\top]\!]^{v,G,\sigma} &= 1\\
[\![\neg\phi]\!]^{v,G,\sigma} &= 1 - [\![\phi]\!]^{v,G,\sigma}\\
[\![\phi_1 \wedge \phi_2]\!]^{v,G,\sigma} &= \min\{[\![\phi_1]\!]^{v,G,\sigma}, [\![\phi_2]\!]^{v,G,\sigma}\}\\
[\![r_1 = r_2]\!]^{v,G,\sigma} &= \begin{cases} 1 & \text{if } \{v' \mid (v,v') \in r_1(G)\} = \{v' \mid (v,v') \in r_2(G)\}\\ 0 & \text{otherwise}\end{cases}\\
[\![I]\!]^{v,G,\sigma} &= \begin{cases} 1 & \text{if } v \text{ is the IRI } I\\ 0 & \text{otherwise}\end{cases}\\
[\![s]\!]^{v,G,\sigma} &= \begin{cases} 1 & \text{if } s \in \sigma(v)\\ 0 & \text{if } \neg s \in \sigma(v)\\ 0.5 & \text{otherwise}\end{cases}\\
[\![\geq_n r.\phi]\!]^{v,G,\sigma} &= \begin{cases} 1 & \text{if } |\{v' \mid (v,v') \in r(G) \text{ and } [\![\phi]\!]^{v',G,\sigma} = 1\}| \geq n\\ 0 & \text{if } |\{v' \mid (v,v') \in r(G)\}|\\ & \quad -|\{v' \mid (v,v') \in r(G) \text{ and } [\![\phi]\!]^{v',G,\sigma} = 0\}| < n\\ 0.5 & \text{otherwise}\end{cases}
\end{aligned}$$

Let $\sum^{G,S}$ be the set of all assignments for graph G and set of shapes S. The "immediate evaluation" operator $\mathbf{T}^{G,S}$ for G and S takes an assignment σ and returns the assignment $\mathbf{T}^{G,S}(\sigma)$ obtained by evaluating the constraint formula ϕ_s for each shape $s \in S$ at each node of G. We write $\mathbf{T}$ instead of $\mathbf{T}^{G,S}$ when G and S are clear from the context.

Definition 4.4.3. [11](Immediate evaluation operator $\mathbf{T}^{G,S}$) The immediate evaluation operator is a function $\mathbf{T}^{G,S} : \sum^{G,S} \to \sum^{G,S}$ defined by $s \in \mathbf{T}^{G,S}(\phi)(v)$ if and only if $[\![\phi_s]\!]^{v,G,\sigma} = 1$, and $s \notin \mathbf{T}^{G,S}(\phi)(v)$ if and only if $[\![\phi_s]\!]^{v,G,\sigma} = 0$.

Let the preorder $\preceq$ over $\sum^{G,S}$ be defined as follows: $\sigma_1 \preceq \sigma_2$ if and only if $\sigma_1(v) \subseteq \sigma_2(v)$ for all v in G. It is defined based on set inclusion, and, therefore, is reflexive, transitive, and antisymmetric. Then $\langle \sum^{G,S}, \preceq \rangle$ is a meet-semilattice as every two elements have a greatest lower bound, the intersection of the two assignments. The immediate evaluation operator is monotonic with respect to $\preceq$ as it preserves set inclusion, i.e. if $\sigma_1 \preceq \sigma_2$ then $\mathbf{T}^{G,S}(\sigma_1) \preceq \mathbf{T}^{G,S}(\sigma_2)$. Then $\mathbf{T}^{G,S}$ emits a (unique) least fixed point over $\sum^{G,S}$ [11], which follows from a weaker version of the Knaster-Tarski theorem [32].

4.4.4 Algorithm

Corman, Reutter and Savković propose a sound approximation algorithm in [11] to decide whether a graph is valid against a set of shapes. The algorithm is parametized by an integer parameter k which limits the Breadth First Search (BFS) depth. If k is bound, then the algorithm is not complete but runs in time polynomial in the size of the graph. If k is unbound, then the algorithm is complete but may run in time exponential in the size of the graph.

The algorithm consists of two steps, the first step consists of computing an assignment matching all constraints enforced by the graph regardless of the target. This is done by computing the least fixed point σ_{minFix} of the immediate evaluation operator $\mathbf{T}^{G,S}$. If the validity of some target node v_0 against shape s cannot be concluded, i.e. if $s \notin \sigma_{\text{minFix}}(v_0)$ and $\neg s \notin \sigma_{\text{minFix}}(v_0)$, then σ_{minFix} is extended by assigning s to v_0 and an attempt is made to propagate constraints from v_0 to its successors, in order for it to satisfy the constraint. In this second step, a constraint satisfying assignment is searched by means of BFS and backtracking.

Example 27. Consider the stratified set of shapes and data graph in Figure 4.1. Suppose one tries to validate v_0 against s_0, hereby requiring a constraint satisfying assignment such that shape s_0 is assigned to v_0. We apply step one of the algorithm in [11].

$$\begin{aligned} \phi_{s_0} &= \geq_1 P.s_1 \\ \phi_{s_1} &= \geq_1 Q.\top \end{aligned} \qquad (4.1)$$

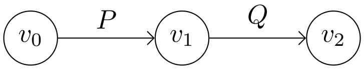


Figure 4.1: A set of shapes and data graph conclusive by $\mathbf{T}$

Initialize $\sigma = \emptyset$

Apply $\mathbf{T}$
Evaluate ϕ_{s_0}

$$\begin{aligned} &[\![\geq_1 P.s_1]\!]^{v_0,G,\sigma} = 0.5 \text{ as } \{v' \mid (v_0, v') \in P(G)\} = \{v_1\} \text{ and } \neg s_1, s_1 \notin \sigma(v_1) \\ &[\![\geq_1 P.s_1]\!]^{v_1,G,\sigma} = 0 \text{ as } \{v' \mid (v_1, v') \in P(G)\} = \emptyset \\ &[\![\geq_1 P.s_1]\!]^{v_2,G,\sigma} = 0 \text{ as } \{v' \mid (v_2, v') \in P(G)\} = \emptyset \end{aligned}$$

Evaluate ϕ_{s_1}

$$\begin{aligned} &[\![\geq_1 Q.\top]\!]^{v_0,G,\sigma} = 0 \text{ as } \{v' \mid (v_0, v') \in Q(G)\} = \emptyset \\ &[\![\geq_1 Q.\top]\!]^{v_1,G,\sigma} = 1 \text{ as } \{v' \mid (v_1, v') \in Q(G)\} = \{v_2\} \text{ and } v_2 \text{ validates } \top \\ &[\![\geq_1 Q.\top]\!]^{v_2,G,\sigma} = 0 \text{ as } \{v' \mid (v_2, v') \in Q(G)\} = \emptyset \end{aligned}$$

Set $\sigma = \{v_0 \to \{\neg s_1\}, v_1 \to \{\neg s_0, s_1\}, v_2 \to \{\neg s_0, \neg s_1\}\}$

Apply $\mathbf{T}$

Evaluate ϕ_{s_0}

$$\llbracket \geq_1 P.s_1 \rrbracket^{v_0,G,\sigma} = 1 \text{ as } \{v' \mid (v_0, v') \in P(G)\} = \{v_1\} \text{ and } s_1 \in \sigma(v_1)$$
$$\llbracket \geq_1 P.s_1 \rrbracket^{v_1,G,\sigma} = 0 \text{ as } \{v' \mid (v_1, v') \in P(G)\} = \emptyset$$
$$\llbracket \geq_1 P.s_1 \rrbracket^{v_2,G,\sigma} = 0 \text{ as } \{v' \mid (v_2, v') \in P(G)\} = \emptyset$$

Evaluate ϕ_{s_1}

$$\llbracket \geq_1 Q.\top \rrbracket^{v_0,G,\sigma} = 0 \text{ as } \{v' \mid (v_0, v') \in Q(G)\} = \emptyset$$
$$\llbracket \geq_1 Q.\top \rrbracket^{v_1,G,\sigma} = 1 \text{ as } \{v' \mid (v_1, v') \in Q(G)\} = \{v_2\} \text{ and } v_2 \text{ validates } \top$$
$$\llbracket \geq_1 Q.\top \rrbracket^{v_2,G,\sigma} = 0 \text{ as } \{v' \mid (v_2, v') \in Q(G)\} = \emptyset$$

Set $\sigma = \{v_0 \rightarrow \{s_0, \neg s_1\}, v_1 \rightarrow \{\neg s_0, s_1\}, v_2 \rightarrow \{\neg s_0, \neg s_1\}\}$

Apply **T**
Evaluate ϕ_{s_0}

$$\llbracket \geq_1 P.s_1 \rrbracket^{v_0,G,\sigma} = 1 \text{ as } \{v' \mid (v_0, v') \in P(G)\} = \{v_1\} \text{ and } s_1 \in \sigma(v_1)$$
$$\llbracket \geq_1 P.s_1 \rrbracket^{v_1,G,\sigma} = 0 \text{ as } \{v' \mid (v_1, v') \in P(G)\} = \emptyset$$
$$\llbracket \geq_1 P.s_1 \rrbracket^{v_2,G,\sigma} = 0 \text{ as } \{v' \mid (v_2, v') \in P(G)\} = \emptyset$$

Evaluate ϕ_{s_1}

$$\llbracket \geq_1 Q.\top \rrbracket^{v_0,G,\sigma} = 0 \text{ as } \{v' \mid (v_0, v') \in Q(G)\} = \emptyset$$
$$\llbracket \geq_1 Q.\top \rrbracket^{v_1,G,\sigma} = 1 \text{ as } \{v' \mid (v_1, v') \in Q(G)\} = \{v_2\} \text{ and } v_2 \text{ validates } \top$$
$$\llbracket \geq_1 Q.\top \rrbracket^{v_2,G,\sigma} = 0 \text{ as } \{v' \mid (v_2, v') \in Q(G)\} = \emptyset$$

Set $\sigma = \{v_0 \rightarrow \{s_0, \neg s_1\}, v_1 \rightarrow \{\neg s_0, s_1\}, v_2 \rightarrow \{\neg s_0, \neg s_1\}\}$

Fixed point reached and $s_0 \in \sigma(v_0)$. △

Example 28. Consider the stratified set of shapes and data graph in Figure 4.2. Suppose one tries to validate v_0 against s_0, hereby requiring a constraint satisfying assignment such that shape s_0 is assigned to v_0. We apply step one of the algorithm in [11].

$$\phi_{s_0} = \geq_1 P.s_0 \qquad (4.2)$$

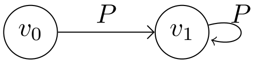


Figure 4.2: A set of shapes and data graph inconclusive by **T**

Initialize $\sigma = \emptyset$

Apply **T**
Evaluate ϕ_{s_0}

$$\llbracket \geq_1 P.s_0 \rrbracket^{v_0,G,\sigma} = 0.5 \text{ as } \{v' \mid (v_0, v') \in P(G)\} = \{v_1\} \text{ and } \neg s_0, s_0 \notin \sigma(v_1)$$
$$\llbracket \geq_1 P.s_0 \rrbracket^{v_1,G,\sigma} = 0.5 \text{ as } \{v' \mid (v_1, v') \in P(G)\} = \{v_1\} \text{ and } \neg s_0, s_0 \notin \sigma(v_1)$$

Set $\sigma = \emptyset$

Apply **T**
Evaluate ϕ_{s_0}

$$\llbracket \geq_1 P.s_0 \rrbracket^{v_0,G,\sigma} = 0.5 \text{ as } \{v' \mid (v_0, v') \in P(G)\} = \{v_1\} \text{ and } \neg s_0, s_0 \notin \sigma(v_1)$$
$$\llbracket \geq_1 P.s_0 \rrbracket^{v_1,G,\sigma} = 0.5 \text{ as } \{v' \mid (v_1, v') \in P(G)\} = \{v_1\} \text{ and } \neg s_0, s_0 \notin \sigma(v_1)$$

Set $\sigma = \emptyset$

Fixed point reached and inconclusive. The algorithm would now continue with step two. △

4.4.5 Complexity

Corman, Reutter and Savković study the computational complexity in [11] of the validation problem for various fragments of SHACL Core. They also study the complexity for a fixed set of shapes and a fixed graph separately, called the data complexity and constraint complexity, respectively.

They show that validation of the fragment $\mathcal{L}$ (SHACL Core) is NP-complete in combined complexity and that this bound is tight for data and graph complexity, even when using stratified negation and just basic operators ($\geq_1, \neg, \wedge$). It is the fragment of $\mathcal{L}$ without property path, counting, and path equality. This fragment is called **stratified $\boldsymbol{\mathcal{L}_{\geq_1,\neg,\wedge}}$** and is defined by the grammar:

$$\phi ::= \top \mid s \mid I \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \geq_1 p.\phi$$

where p is an IRI.

Example 29. Consider the stratified set of shapes, its dependency graph, and the data graph in Figure 4.3. Suppose one tries to validate v_0 against s_0, hereby requiring a constraint satisfying assignment such that shape s_0 is assigned to v_0.

$$\begin{aligned}\phi_{s_0} &= \geq_1 P.s_1 \wedge \geq_1 P.s_2\\ \phi_{s_1} &= \neg \geq_1 P.s_2\\ \phi_{s_2} &= \geq_1 P.s_2\end{aligned}$$

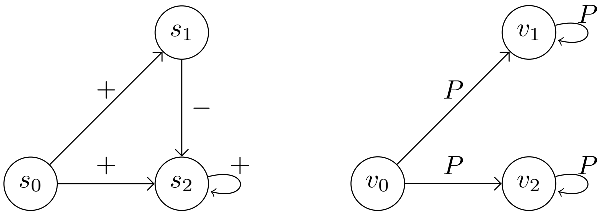


Figure 4.3: A stratified set of shapes, its dependency graph, and a data graph

Note that the set of shapes is stratified, and suppose it has the following mapping: $\alpha = \{s_2 \to 0, s_0 \to 1, s_1 \to 1\}$. Now S can be partitioned into the following distinct but semantically equivalent stratifications (ordered sequence of strata): $\langle\{s_2\},\{s_0,s_1\}\rangle, \langle\{s_2\},\{s_0\},\{s_1\}\rangle, \langle\{s_2\},\{s_1\},\{s_0\}\rangle$. Starting at the lowest strata ensures that negated shapes are processed before the shapes that reference them.

Utilizing this, we start with the stratum $\{s_2\}$. We evaluate ϕ_{s_2} at each node, requiring a node to reach a node satisfying s_2 over P. Shape s_2 can clearly be assigned to v_1 and v_2 due to self loops and then to v_0. Now we reach the stratum $\{s_0, s_1\}$. We evaluate ϕ_{s_0} at each node, requiring a node to reach a node satisfying s_1 and a node to reach a node satisfying s_2 over P. Shape s_1 requires a node to not reach a node satisfying s_2 over P. It is clear that s_1 cannot be assigned to any node as v_1 and v_2 assign s_2. Now one needs to backtrack and undo the assignment of s_2 to v_1 or v_2. This backtracking behavior is in the worst case exponential in the size of the graph. Hence the intractability. △

4.4.6 Towards Tractability

State of the Art

Corman, Reutter and Savković show in [11] that allowing disjunction as a native operator, i.e. writing $\phi_1 \vee \phi_2$ instead of $\neg(\neg\phi_1 \wedge \neg\phi_2)$, and disallowing negation is sufficient to gain tractability. This fragment is called $\boldsymbol{\mathcal{L}_{\geq_n,\wedge,\vee,r,\mathrm{EQ}}}$ and is defined by the grammar:

$$\phi ::= \top \mid s \mid I \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \geq_n r.\phi \mid \mathrm{EQ}(r_1, r_2)$$

They also show that validation of the sub-fragment $\boldsymbol{\mathcal{L}_{\geq_n,\wedge,\vee}}$ of $\mathcal{L}_{\geq_n,\wedge,\vee,r,\mathrm{EQ}}$ without property path and path equality is P-hard in combined complexity. Therefore, validation of $\mathcal{L}_{\geq_n,\wedge,\vee,r,\mathrm{EQ}}$ is P-complete in combined complexity [11].

An alternative approach to gain tractability, while supporting all SHACL Core operators, is to strengthen the stratification condition, called strict stratification [10]. Let this fragment be called

strictly stratified $\mathcal{L}$. Its validation is P-complete in data complexity [13] and in P in combined complexity [10], and, therefore, P-complete in combined complexity.

Definition 4.4.4. [10](Strict stratification) A set of shapes S is strictly stratified if, for any pair (s_1, s_2) of nodes in its positive contracted dependency graph, either:

i there is at most one path from s_1 to s_2, or

ii all paths from s_1 to s_2 are positive.

Remark. If there exists a negative path from s_1 to s_2, then this is the only path from s_1 to s_2.

Example 30. The shapes in shapes graph G of Example 22 are not strictly stratified as its positive contracted dependency graph (see Example 24) has two paths from *ex:HumanShape* to *ex:FatherShape* of which one path is negative. △

Proposition 4.4.2. *A strictly stratified set of shapes S is stratified.*

Proof. Suppose we have a strictly stratified set of shapes S. By Definition 4.3.3, no negative cycle exists in the positive contracted dependency graph. In order to derive a contradiction, suppose S is not stratified. Then by Proposition 4.4.1, the dependency graph contains a cycle containing a negative edge. By Definition 4.3.3, this negative cycle exists in the positive contracted dependency graph as only strongly connected components consisting of only positives edges are contracted. This is a contradiction, so S must be stratified if it is strictly stratified. □

Intractability does not hold for strictly stratified shapes, which intuitively guarantees that no backtracking is needed. In that case, step two of the algorithm in [11] is not needed and the validity can be determined in polynomial time in $|G| + |S|$. We consider a shape s with a single target node v_0. If $s \in \sigma_{\text{minFix}}(v_0)$, then the graph is valid as we found a minimal fixed-point assignment assigning s to v_0. If $\neg s \in \sigma_{\text{minFix}}(v_0)$, then the graph is invalid as we found a minimal fixed-point assignment assigning $\neg s$ to v_0. Due to monotonicity of $\mathbf{T}$, every other fixed point must extend it, hence no fixed-point assignment for $\mathbf{T}$ exists assigning s to v_0. Else, i.e. $\neg s, s \notin \sigma_{\text{minFix}}(v_0)$, then a constraint satisfying assignment σ' must exist, meaning that $s \in \sigma'(v)$ implies $[\![s]\!]^{v,G,\sigma'} = 1$ and $\neg s \in \sigma'(v)$ implies $[\![s]\!]^{v,G,\sigma'} = 0$, such that $s \in \sigma'(v_0)$, hence the graph is valid.

A New Tractable Fragment

Allowing disjunction and universal quantification as a native operator, i.e. writing $\phi_1 \vee \phi_2$ and $\forall r.\phi$ instead of $\neg(\neg\phi_1 \wedge \neg\phi_2)$ and $\neg(\geq_1 r.\neg\phi)$, respectively, results in a more expressive fragment by allowing universal quantification and disjunction to be expressed without the use of negation. This implies that their use is not constrained by the strict stratification restrictions. We call this new fragment **strictly stratified $\mathcal{L}^+$** and it is defined by the grammar:

$\phi ::= \top \mid s \mid I \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \geq_n r.\phi \mid \mathrm{EQ}(r_1, r_2) \mid \phi_1 \vee \phi_2 \mid \forall r.\phi$

The mapping from this grammar to SHACL is an extension on the mapping from $\mathcal{L}$ to SHACL [11] in which:

- $\phi_s = \forall r.\phi$ maps to a node shape s which declares a constraint of kind *PropertyConstraintComponent* with as value (for mandatory property *sh:property*) a property shape that has SPARQL property path r mapped to a SHACL property path r' as value for *sh:path*, and a constraint of kind *NodeConstraintComponent* with as value (for mandatory property *sh:node*) constraint ϕ mapped to node shape s':

```
s
  a sh:NodeShape ;
  sh:property [
    sh:path r′ ;
    sh:node s′
  ] .
```

- $\phi_s = \phi_1 \lor \phi_2$ maps to a node shape s which declares a constraint of kind *OrConstraintComponent* with as value (for mandatory property *sh:or*) a list with the constraint ϕ_1 and ϕ_2 mapped to node shape s'_1 and s'_2, respectively, as its members:

```
s
  a sh:NodeShape ;
  sh:or ( s'_1 s'_2 ) .
```

We show that the fragments $\mathcal{L}_{\geq_n,\land,\lor,r,\mathrm{EQ}}$ and strictly stratified $\mathcal{L}$ are included in strictly stratified $\mathcal{L}^+$.

Theorem 1. *Every set of shapes defined in* $\mathcal{L}_{\geq_n,\land,\lor,r,EQ}$ *and strictly stratified* $\mathcal{L}$ *can be defined in strictly stratified* $\mathcal{L}^+$.

Proof. All operators of $\mathcal{L}_{\geq_n,\land,\lor,r,\mathrm{EQ}}$ and strictly stratified $\mathcal{L}$ are natively supported in strictly stratified $\mathcal{L}^+$. So any set of shapes defined in these fragments can be defined in the grammar of strictly stratified $\mathcal{L}^+$. It remains to show that the set of shapes is strictly stratified.

By definition, any set of shapes defined in strictly stratified $\mathcal{L}$ must be strictly stratified. The fragment $\mathcal{L}_{\geq_n,\land,\lor,r,\mathrm{EQ}}$ has no negation operator, so every path in the positive contracted dependency graph for a set of shapes defined in this fragment must be positive. So any set of shapes defined in $\mathcal{L}_{\geq_n,\land,\lor,r,\mathrm{EQ}}$ must also be strictly stratified. □

The validation of strictly stratified $\mathcal{L}^+$ remains in P in combined complexity and is P-hard due to inclusion of $\boldsymbol{\mathcal{L}_{\geq_n,\land,\lor}}$ (used for showing P-hardness in [11]), and, therefore, is P-complete in combined complexity.

Theorem 2. *If* $\neg s \notin \sigma_{minFix}(v_0)$ *for a set of shapes defined in strictly stratified* $\mathcal{L}^+$*, then an assignment* σ *exists such that (a)* $s \in \sigma(v_0)$ *and (b)* $\sigma \preceq \boldsymbol{T}(\sigma)$.

Proof. [1] We abuse notation by writing $s(v) \in \sigma$ instead of $s \in \sigma(v)$ (similarly for $\neg s$). The premise results in two cases, either $s(v_0) \in \sigma_{\mathrm{minFix}}$, and then σ_{minFix} trivially satisfies condition (a) and (b), or $\{\neg s(v_0), s(v_0)\} \cap \sigma_{\mathrm{minFix}} = \emptyset$, for which we prove the existence of σ satisfying condition (a) and (b).

Let S be the set of shapes defined in strictly stratified $\mathcal{L}^+$ that is normalized, i.e. constraint definitions without nested expressions. A normalized set of shapes can be obtained by introducing a fresh shape for each subexpression. This transformation can be performed in polynomial time and results in an equivalent set of shapes whose constraints contain at most one operator [11].

Let P_S be the dependency graph of S, then S can be partitioned in a stratification $\langle S_1, ..., S_n\rangle$ such that:

i If $s_1, s_2 \in S_i$, then there is no negative path in P_S between s_1 and s_2.

ii If $s_1 \in S_i$ and $s_2 \in S_{i+1}$, then there is at most one negative path in P_S from s_2 to s_1 and no other path between s_1 and s_2.

iii Else, i.e. $s_1 \in S_i$ and $s_2 \in S_j$ with $i \neq j$, $i \neq j+1$, and $j \neq i+1$, there exists no path in P_S between s_1 and s_2.

Let $A_1, ..., A_n$ be the corresponding set of atoms of the form $s(v)$ where $s(v) \in A_i$ if and only if: $s \in S_i$, $v \in G$, and $s(v)$ has no value assigned, i.e. $\{\neg s(v), s(v)\} \cap \sigma_{\mathrm{minFix}} = \emptyset$.

For each $1 \leq i \leq n$ we show there must exist two assignments σ_i^+ and σ_i^- such that:

i σ_i^+ and σ_i^- satisfy condition (b).

ii for each $s(v) \in A_i$, $s(v) \in \sigma_i^+$.

iii for each $s(v) \in A_i$, $\neg s(v) \in \sigma_i^-$.

[1] I would like to thank Julien Corman for his help with this proof

In particular, let $1 \leq k \leq n$ be the index of the stratum S_k containing the target, i.e. $s_0(v_0) \in A_k$. Then assignment σ_k^+ satisfies condition (b) by (i) and condition (a) by (ii) since $s_0(v_0) \in A_k$ and thus $s_0(v_0) \in \sigma_k^+$.

We define σ_i^+ and σ_i^- inductively as follows:

$$\begin{aligned}\sigma_1^+ &= \sigma_{\text{minFix}} \cup A_1\\ \sigma_i^+ &= \sigma_{i-1}^- \cup A_i\end{aligned}$$

$$\begin{aligned}\sigma_1^- &= \sigma_{\text{minFix}} \cup \{\neg s(v) \mid s(v) \in A_1\}\\ \sigma_i^- &= \sigma_{i-1}^+ \cup \{\neg s(v) \mid s(v) \in A_i\}\end{aligned}$$

By construction (ii) and (iii) are satisfied, it remains to show that (i) is satisfied, i.e. that the two assignments satisfy condition (b). We apply proof by induction on i for σ_i^+ while leaving out the identical proof for σ_i^-.

- Base case $i = 1$: $\sigma_1^+ \subseteq \mathbf{T}(\sigma_1^+)$

  We have:

  i $\sigma_{\text{minFix}} \subseteq \mathbf{T}(\sigma_{\text{minFix}})$ because σ_{minFix} is a fixed-point of $\mathbf{T}$.

  ii $\sigma_{\text{minFix}} \subseteq \sigma_1^+$ from the definition.

  iii $\mathbf{T}(\sigma_{\text{minFix}}) \subseteq \mathbf{T}(\sigma_1^+)$ from (ii) and the monotonicity of $\mathbf{T}$.

  iv $\sigma_{\text{minFix}} \subseteq \mathbf{T}(\sigma_1^+)$ from (i) and (iii).

  By the definitions, for any $s(v)$ or $\neg s(v) \in \sigma_1^+$, either:

  1 $s(v) \in \sigma_{\text{minFix}}$: by (iv), $s(v) \in \mathbf{T}(\sigma_1^+)$.

  2 $\neg s(v) \in \sigma_{\text{minFix}}$: by (iv), $\neg s(v) \in \mathbf{T}(\sigma_1^+)$.

  3 $s(v) \in A_1$: we show that $s(v) \in \mathbf{T}(\sigma_1^+)$, i.e. $[\![\phi_s]\!]^{v,G,\sigma_1^+} = 1$, where ϕ_s is the constraint associated with s.

  We consider all possible syntactic forms of ϕ_s:

  - $\phi_s = \top$.
    For any assignment σ, $[\![\top]\!]^{v,G,\sigma} = 1$ must hold.
    In particular, $[\![\top]\!]^{v,G,\sigma_1^+} = 1$ must hold.
    Therefore $[\![\phi_s]\!]^{v,G,\sigma_1^+} = 1$.
  - $\phi_s = s'$.
    From the definition of A_1, $\{s(v), \neg s(v)\} \cap \sigma_{\text{minFix}} = \emptyset$.
    So $[\![s]\!]^{v,G,\sigma_{\text{minFix}}} = 0.5$.
    Then because σ_{minFix} is a fixed-point of $\mathbf{T}$, $\sigma_{\text{minFix}} = \mathbf{T}(\sigma_{\text{minFix}})$ must hold.
    So from the definition of $\mathbf{T}$, $[\![\phi_s]\!]^{v,G,\sigma_{\text{minFix}}} = [\![s]\!]^{v,G,\sigma_{\text{minFix}}}$.
    Therefore $[\![\phi_s]\!]^{v,G,\sigma_{\text{minFix}}} = 0.5$ must hold.
    Then because $\phi_s = s'$, $[\![s']\!]^{v,G,\sigma_{\text{minFix}}} = 0.5$ must hold.
    So:
    $$\{\neg s'(v), s'(v)\} \cap \sigma_{\text{minFix}} = \emptyset \tag{4.3}$$
    In addition, because S is stratified and $s \in S_1$, $s' \in S_1$ must hold.
    So from 4.3:
    $$s'(v) \in A_1 \tag{4.4}$$
    So from 4.3, 4.4, and the definition of σ_1^+, $s'(v) \in \sigma_1^+$.
    Therefore $[\![s']\!]^{v,G,\sigma_1^+} = 1$.
    So $[\![\phi_s]\!]^{v,G,\sigma_1^+} = [\![s']\!]^{v,G,\sigma_1^+} = 1$.

- $\phi_s = s_1 \wedge s_2$.
  Similarly to the proof for $\phi_s = s'$, $[\![s_1 \wedge s_2]\!]^{v,G,\sigma_{\text{minFix}}} = 0.5$ must hold.
  Then either:
  - $$\{\neg s_1(v), s_1(v), \neg s_2(v), s_2(v)\} \cap \sigma_{\text{minFix}} = \emptyset \tag{4.5}$$
    In addition, because S is stratified and $s \in S_1$, $s_1, s_2 \in S_1$ must hold.
    So from 4.5:
    $$s_1(v) \in A_1 \text{ and } s_2(v) \in A_1 \tag{4.6}$$
  - $$\{\neg s_1(v), s_1(v), \neg s_2(v), s_2(v)\} \cap \sigma_{\text{minFix}} = \{s_1(v)\}$$
    Similarly to the first case:
    $$s_1(v) \in \sigma_{\text{minFix}} \text{ and } s_2(v) \in A_1 \tag{4.7}$$
  - $$\{\neg s_1(v), s_1(v), \neg s_2(v), s_2(v)\} \cap \sigma_{\text{minFix}} = \{s_2(v)\}$$
    Similarly to the first case:
    $$s_1(v) \in A_1 \text{ and } s_2(v) \in \sigma_{\text{minFix}} \tag{4.8}$$

  So for any case, from 4.6, 4.7, or 4.8, and the definition of σ_1^+, $s_1(v), s_2(v) \in \sigma_1^+$.
  Therefore $[\![s_1]\!]^{v,G,\sigma_1^+} = [\![s_2]\!]^{v,G,\sigma_1^+} = 1$.
  So $[\![\phi_s]\!]^{v,G,\sigma_1^+} = [\![s_1 \wedge s_2]\!]^{v,G,\sigma_1^+} = 1$.
- $\phi_s = s_1 \vee s_2$.
  Similarly to the proof for $\phi_s = s'$, $[\![s_1 \vee s_2]\!]^{v,G,\sigma_{\text{minFix}}} = [\![\neg(\neg s_1 \wedge \neg s_2)]\!]^{v,G,\sigma_{\text{minFix}}} = 0.5$ must hold.
  Then either:
  - $$\{\neg s_1(v), s_1(v), \neg s_2(v), s_2(v)\} \cap \sigma_{\text{minFix}} = \emptyset \tag{4.9}$$
    In addition, because S is stratified and $s \in S_1$, $s_1, s_2 \in S_1$ must hold.
    So from 4.9:
    $$s_1(v) \in A_1 \text{ and } s_2(v) \in A_1 \tag{4.10}$$
    So from 4.9, 4.10, and the definition of σ_1^+, $s_1(v), s_2(v) \in \sigma_1^+$.
    Therefore $[\![s_1]\!]^{v,G,\sigma_1^+} = [\![s_2]\!]^{v,G,\sigma_1^+} = 1$.
  - $$\{\neg s_1(v), s_1(v), \neg s_2(v), s_2(v)\} \cap \sigma_{\text{minFix}} = \{\neg s_1(v)\} \tag{4.11}$$
    Similarly to the first case:
    $$\neg s_1(v) \in \sigma_{\text{minFix}} \text{ and } s_2(v) \in A_1 \tag{4.12}$$
    So from 4.11, 4.12, and the definition of σ_1^+, $\neg s_1(v), s_2(v) \in \sigma_1^+$.
    Therefore $[\![s_1]\!]^{v,G,\sigma_1^+} = 0$ and $[\![s_2]\!]^{v,G,\sigma_1^+} = 1$.
  - $$\{\neg s_1(v), s_1(v), \neg s_2(v), s_2(v)\} \cap \sigma_{\text{minFix}} = \{\neg s_2(v)\}$$
    This proof is almost identical to the previous case.

  So for any case, $[\![\phi_s]\!]^{v,G,\sigma_1^+} = [\![s_1 \vee s_2]\!]^{v,G,\sigma_1^+} = [\![\neg(\neg s_1 \wedge \neg s_2)]\!]^{v,G,\sigma_1^+} = 1$.

- $\phi_s = \geq_n r.s'$.
  Similarly to the proof for $\phi_s = s'$, $[\![\geq_n r.s']\!]^{v,G,\sigma_{\text{minFix}}} = 0.5$ must hold.
  So there must be $\{v_1, .., v_n\}$ such that:

$$(v, r, v_j) \in G \text{ for each } j \in \{1..n\} \tag{4.13}$$
$$\neg s'(v_j) \notin \sigma_{\text{minFix}} \text{ for each } j \in \{1..n\} \tag{4.14}$$
$$s'(v_j) \notin \sigma_{\text{minFix}} \text{ for some } j \in \{1..n\}$$

  Then either, for each $j \in \{1..n\}$:

  *
$$\{\neg s'(v_j), s'(v_j)\} \cap \sigma_{\text{minFix}} = \emptyset \tag{4.15}$$

    In addition, because S is stratified and $s \in S_1$, $s' \in S_1$ must hold.
    So from 4.15:
$$s'(v_j) \in A_1 \tag{4.16}$$

  *
$$\{\neg s'(v_j), s'(v_j)\} \cap \sigma_{\text{minFix}} = \{s'(v_j)\} \tag{4.17}$$

    So from 4.17:
$$s'(v_j) \in \sigma_{\text{minFix}} \tag{4.18}$$

  So for any case, from 4.16 or 4.18, and the definition of σ_1^+, for each $j \in \{1..n\}$:

$$s'(v_j) \in \sigma_1^+ \tag{4.19}$$

  Therefore from 4.13 and 4.19, $[\![\geq_n r.s']\!]^{v,G,\sigma_1^+} = 1$.
  So $[\![\phi_s]\!]^{v,G,\sigma_1^+} = [\![\geq_n r.s']\!]^{v,G,\sigma_1^+} = 1$.
- $\phi_s = \forall r.s'$.
  Similarly to the proof for $\phi_s = s'$, $[\![\forall r.s']\!]^{v,G,\sigma_{\text{minFix}}} = [\![\neg(\geq_1 r.\neg s')]\!]^{v,G,\sigma_{\text{minFix}}} = 0.5$ must hold.
  So there must be $\{v_1, .., v_n\}$ for $n = |\{v' \mid (v, v') \in r(G)\}|$ such that:

$$(v, r, v_j) \in G \text{ for all } j \in \{1..n\} \tag{4.20}$$
$$\neg s'(v_j) \notin \sigma_{\text{minFix}} \text{ for all } j \in \{1..n\} \tag{4.21}$$
$$s'(v_j) \notin \sigma_{\text{minFix}} \text{ for some } j \in \{1..n\}$$

  Similarly to the proof for $\phi_s = \geq_n r.s'$, for each $j \in \{1..n\}$:

$$s'(v_j) \in \sigma_1^+ \tag{4.22}$$

  Therefore from 4.20 and 4.22, $[\![\geq_1 r.\neg s']\!]^{v,G,\sigma_1^+} = 0$.
  So $[\![\phi_s]\!]^{v,G,\sigma_1^+} = [\![\forall r.s']\!]^{v,G,\sigma_1^+} = [\![\neg(\geq_1 r.\neg s')]\!]^{v,G,\sigma_1^+} = 1$.

The operator $\neg$ cannot occur because S is stratified and $i = 1$. The other operators cannot occur as they are independent of σ, therefore $[\![s]\!]^{v,G,\sigma_{\text{minFix}}} \neq 0.5$, so $s(v) \notin A_1$.

- Inductive case $i > 1$: $\sigma_i^+ \subseteq \mathbf{T}(\sigma_i^+)$

  We have:

  i $\sigma_{i-1}^- \subseteq \mathbf{T}(\sigma_{i-1}^-)$ by the induction hypothesis.

  ii $\sigma_{i-1}^- \subseteq \sigma_i^+$ from the definition.

  iii $\mathbf{T}(\sigma_{i-1}^-) \subseteq \mathbf{T}(\sigma_i^+)$ from (ii) and the monotonicity of $\mathbf{T}$.

  iv $\sigma_{i-1}^- \subseteq \mathbf{T}(\sigma_i^+)$ from (i) and (iii).

  By the definitions, for any $s(v)$ or $\neg s(v) \in \sigma_i^+$, either:

1 $s(v) \in \sigma^-_{i-1}$: by (iv), $s(v) \in \mathbf{T}(\sigma^+_i)$.

2 $\neg s(v) \in \sigma^-_{i-1}$: by (iv), $\neg s(v) \in \mathbf{T}(\sigma^+_i)$.

3 $s(v) \in A_i$: we show that $s(v) \in \mathbf{T}(\sigma^+_i)$, i.e. $[\![\phi_s]\!]^{v,G,\sigma^+_i} = 1$, where ϕ_s is the constraint associated with s.

We consider all possible syntactic forms of ϕ_s:

- $\phi_s = \top$.
  This proof is identical to the base case proof.
- $\phi_s = \neg s'$.
  Similarly to the base case proof for $\phi_s = s'$, $[\![\neg s']\!]^{v,G,\sigma_{\text{minFix}}} = 0.5$ must hold.
  So:
  $$\{s'(v), \neg s'(v)\} \cap \sigma_{\text{minFix}} = \emptyset \tag{4.23}$$
  In addition, because S is stratified and $s \in S_i$, $s' \in S_{i-1}$ must hold.
  So from 4.23:
  $$s'(v) \in A_{i-1} \tag{4.24}$$
  So from 4.23, 4.24, and the definition of σ^-_{i-1}:
  $$\neg s'(v) \in \sigma^-_{i-1} \tag{4.25}$$
  So from 4.23, 4.25 and the definition of σ^+_i, $\neg s'(v) \in \sigma^+_i$.
  Therefore $[\![s']\!]^{v,G,\sigma^+_i} = 0$.
  So $[\![\phi_s]\!]^{v,G,\sigma^+_i} = [\![\neg s']\!]^{v,G,\sigma^+_i} = 1$.
- $\phi_s = s'$.
  Similarly to the base case proof, $[\![s']\!]^{v,G,\sigma_{\text{minFix}}} = 0.5$ must hold.
  So:
  $$\{\neg s'(v), s'(v)\} \cap \sigma_{\text{minFix}} = \emptyset \tag{4.26}$$
  In addition, because S is stratified and $s \in S_i$, $s' \in S_i$ must hold.
  So from 4.26:
  $$s'(v) \in A_i \tag{4.27}$$
  So from 4.26, 4.27, and the definition of σ^+_i, $s'(v) \in \sigma^+_i$.
  Therefore $[\![s']\!]^{v,G,\sigma^+_i} = 1$.
  So $[\![\phi_s]\!]^{v,G,\sigma^+_i} = [\![s']\!]^{v,G,\sigma^+_i} = 1$.
- $\phi_s = s_1 \wedge s_2$.
  This proof is almost identical to the base case proof.
- $\phi_s = s_1 \vee s_2$.
  This proof is almost identical to the base case proof.
- $\phi_s = \geq_n r.s'$.
  This proof is almost identical to the base case proof.
- $\phi_s = \forall r.s'$.
  This proof is almost identical to the base case proof.

The other operators cannot occur as they are independent of σ, therefore $[\![s]\!]^{v,G,\sigma_{\text{minFix}}} \neq 0.5$, so $s(v) \notin A_i$. Proofs that are almost identical to the base case proof use A_i instead of A_1 and σ^+_i instead of σ^+_1.

□

Example 31. We continue with Example 29 and provide an example to Theorem 2. Firstly, we show that the stratified set of shapes is not strictly stratified, secondly, we make the set of shapes strictly stratified, thirdly, we argue that executing the fixed-point algorithm on the given data graph remains inconclusive, and lastly, we show the existence of a constraint satisfying assignment that successfully assigns the shape. Hereby concluding the graph's validity with respect to the strictly stratified set of shapes.

$$\begin{aligned}\phi_{s_0} &= \geq_1 P.s_1 \wedge \geq_1 P.s_2\\ \phi_{s_1} &= \neg \geq_1 P.s_2\\ \phi_{s_2} &= \geq_1 P.s_2\end{aligned}$$

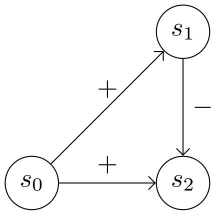


Figure 4.4: A stratified set of shapes and its positive contracted dependency graph

The strict stratification conditions do not hold as there exist two paths from node s_0 to s_2 of which one is negative (see Figure 4.4). We make the set of shapes strictly stratified by removing the direct dependency of s_0 on s_2 (see Figure 4.5). Note that the stratified and strictly stratified set of shapes are not equivalent.

$$\begin{aligned}\phi_{s_0} &= \geq_1 P.s_1\\ \phi_{s_1} &= \neg \geq_1 P.s_2\\ \phi_{s_2} &= \geq_1 P.s_2\end{aligned}$$

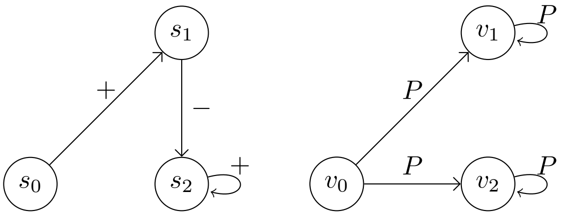


Figure 4.5: A strictly stratified set of shapes, its dependency graph, and a data graph

All constraints depend on shape s_2, therefore, by the semantics of $[\![\geq_n r.\phi]\!]^{v,G,\sigma}$, no true value can be assigned unless s_2 is assigned. A false value won't be assigned either as each node has at least one path with label P. So the unknown value will be assigned, hence the fixed-point algorithm remains inconclusive (see Example 28 for a detailed example).

By Theorem 2, we conclude that the data graph is valid with respect to the strictly stratified set of shapes. We now show the existence of a constraint satisfying assignment assigning s_0 to target node v_0.

We partition S according to the strict stratification conditions into the following stratification: $\langle\{s_2\},\{s_0,s_1\}\rangle$. Now (i) is satisfied as no negative paths exist between the nodes of each stratum; (ii) is satisfied as (at most) one negative path exists from node s_0 to s_2 and s_1 to s_2, and no other paths between each node pair; and lastly, (iii) is trivially satisfied.

We calculate σ_2^+ because $s_0 \in S_2$, using the definitions in the proof for Theorem 2:

$$\begin{aligned}\sigma_1^- =&\emptyset \cup \{\neg s_2(v_0), \neg s_2(v_1), \neg s_2(v_2)\}\\ \sigma_2^+ =&\{\neg s_2(v_0), \neg s_2(v_1), \neg s_2(v_2)\}\\ &\cup \{s_0(v_0), s_0(v_1), s_0(v_2), s_1(v_0), s_1(v_1), s_1(v_2)\}\end{aligned}$$

The assignment σ_2^+ successfully assigns s_0 to v_0. It remains to show that $\sigma_2^+ \preceq \mathbf{T}(\sigma_2^+)$:

Apply **T**
Evaluate ϕ_{s_0}

$$\begin{aligned}
\llbracket \geq_1 P.s_1 \rrbracket^{v_0,G,\sigma_2^+} &= 1 \text{ as } \{v' \mid (v_0, v') \in P(G)\} = \{v_1, v_2\} \text{ and } s_1(v_1) \in \sigma_2^+ \\
\llbracket \geq_1 P.s_1 \rrbracket^{v_1,G,\sigma_2^+} &= 1 \text{ as } \{v' \mid (v_1, v') \in P(G)\} = \{v_1\} \text{ and } s_1(v_1) \in \sigma_2^+ \\
\llbracket \geq_1 P.s_1 \rrbracket^{v_2,G,\sigma_2^+} &= 1 \text{ as } \{v' \mid (v_2, v') \in P(G)\} = \{v_2\} \text{ and } s_1(v_2) \in \sigma_2^+
\end{aligned} \tag{4.28}$$

Evaluate ϕ_{s_1}

$$\begin{aligned}
\llbracket \neg \geq_1 P.s_2 \rrbracket^{v_0,G,\sigma_2^+} &= 1 \text{ as } \{v' \mid (v_0, v') \in P(G)\} = \{v_1, v_2\} \text{ and } \neg s_2(v_1) \in \sigma_2^+ \\
\llbracket \neg \geq_1 P.s_2 \rrbracket^{v_1,G,\sigma_2^+} &= 1 \text{ as } \{v' \mid (v_1, v') \in P(G)\} = \{v_1\} \text{ and } \neg s_2(v_1) \in \sigma_2^+ \\
\llbracket \neg \geq_1 P.s_2 \rrbracket^{v_2,G,\sigma_2^+} &= 1 \text{ as } \{v' \mid (v_2, v') \in P(G)\} = \{v_2\} \text{ and } \neg s_2(v_2) \in \sigma_2^+
\end{aligned} \tag{4.29}$$

Evaluate ϕ_{s_2}

$$\begin{aligned}
\llbracket \geq_1 P.s_2 \rrbracket^{v_0,G,\sigma_2^+} &= 0 \text{ as } \{v' \mid (v_0, v') \in P(G)\} = \{v_1, v_2\} \text{ and } \neg s_2(v_1) \in \sigma_2^+ \\
\llbracket \geq_1 P.s_2 \rrbracket^{v_1,G,\sigma_2^+} &= 0 \text{ as } \{v' \mid (v_1, v') \in P(G)\} = \{v_1\} \text{ and } \neg s_2(v_1) \in \sigma_2^+ \\
\llbracket \geq_1 P.s_2 \rrbracket^{v_2,G,\sigma_2^+} &= 0 \text{ as } \{v' \mid (v_2, v') \in P(G)\} = \{v_2\} \text{ and } \neg s_2(v_2) \in \sigma_2^+
\end{aligned} \tag{4.30}$$

So $\mathbf{T}(\sigma_2^+) = \{s_0(v_0), s_0(v_1), s_0(v_2), s_1(v_0), s_1(v_1), s_1(v_2), \neg s_2(v_0), \neg s_2(v_1), \neg s_2(v_2)\}$.
Thus, $\sigma_2^+ = \mathbf{T}(\sigma_2^+)$. △

4.5 Conclusions

Shapes are RDF resources that declare constraints using the parameters of constraint components. Shapes can reference other shapes by means of referencing constraints which validate nodes against other shapes without being targeted directly by them. The dependency graph is the labeled graph with a directed edge for each shape reference labeled with its parity. Stratification and strict stratification impose restrictions on the cycles and parities of edges in the dependency graph. A shape is recursive if the dependency graph contains a cycle of which this shape is part of.

The validation of recursive SHACL is based on a minimal fixed-point assignment. The assignment is partial such that it is possible to neither assign a shape nor its negation to a node. Validation of $\mathcal{L}$ (SHACL Core) is intractable, even when using the severely limited fragment stratified $\mathcal{L}_{\geq_1,\neg,\wedge}$ with stratified negation and just basic operators. Table 4.3 summarizes all identified recursive SHACL fragments.

Table 4.3: Combined complexity of validation and supported operators of identified recursive SHACL fragments

The operators $\top$, s, I are supported by any fragment. The columns r and p denote support for property and predicate path, respectively. -c stands for complete.

Fragment	**Complexity**	$\geq_n$	$\geq_1$	$\neg$	$\wedge$	$\vee$	r	p	EQ	$\forall$
$\mathcal{L}$ (SHACL Core)	NP-c	X	X	X	X		X	X	X	
stratified $\mathcal{L}_{\geq_1,\neg,\wedge}$	NP-c		X	X	X			X		
$\mathcal{L}_{\geq_n,\wedge,\vee,r,\mathrm{EQ}}$	P-c	X	X		X	X	X	X	X	
strictly stratified $\mathcal{L}$	P-c	X	X	X	X		X	X	X	
strictly stratified $\mathcal{L}^+$	P-c	X	X	X	X	X	X	X	X	X

So far three tractable recursive SHACL fragments have been identified: $\mathcal{L}_{\geq_n,\wedge,\vee,r,\mathrm{EQ}}$ which has an additional native operator for disjunction but disallows negation, introduced in [11]; strictly stratified $\mathcal{L}$ with strictly stratified negation, introduced in [10]; and strictly stratified $\mathcal{L}^+$ with strictly stratified negation and additional native operators for universal quantification and disjunction, introduced in this thesis.

Our new fragment strictly stratified $\mathcal{L}^+$ includes $\mathcal{L}_{\geq_n,\wedge,\vee,r,\mathrm{EQ}}$ and strictly stratified $\mathcal{L}$, and allows for additional constraints to be expressed using native operators to express universal quantification and disjunction without the use of negation. This implies that use of such constraints is not constrained by the strict stratification restrictions.

Validation of these fragments is tractable as a polynomial procedure exists deciding whether an assignment σ verifying $s \in \sigma(v_0)$ and $\sigma \preceq \mathbf{T}(\sigma)$ exists, e.g. computing the least fixed point σ_{minFix} of the immediate evaluation operator $\mathbf{T}$. Then, a target node v_0 is valid with respect to a shape s if and only if $\neg s \notin \sigma(v_0)$. We formally proved this property for our new fragment strictly stratified $\mathcal{L}^+$, hereby proving that validation of all tractable recursive fragments identified so far is in P in combined complexity.

Chapter 5

SPARQL-based Validation vs. Native Implementation

5.1 Introduction

The previous chapter discusses the theoretical aspects to validating SHACL and how validation is based on the validity of nodes against shapes. Shapes declare constraints and a node is valid against a shape if it is valid against the constraints declared by this shape. This chapter takes a practical view towards validating non-referencing constraints, i.e. the constraints whose validation does not depend on other shapes, and focuses on the differences between SPARQL-based validation and a native implementation for validating these constraints.

The DASH Data Shapes Vocabulary (DASH)[21] is a collection of reusable extensions to SHACL. It adds additional SHACL constraint and target types, components for representing test cases, suggestions to fix constraint violations, and an extended validation results vocabulary. DASH serves as a reference implementation of SHACL by providing default validators in SPARQL and JavaScript code.

We focus on the SHACL constraint components that have a SPARQL-based constraint validator defined in the DASH namespace. These are all non-referencing constraint components. In order to understand SPARQL-based evaluation, we investigate the SPARQL queries used to validate these constraints by diving into query plans generated by Amazon Neptune[1] for sample instances. We would like to clarify that the goal is purely to understand and assess SPARQL-based validation and not the engine itself. The generated query plans may differ among data sets and engine versions.

Based on the intended semantics and query plans we come up with a potential native implementation using a repository API in pseudo code. The repository API follows an iterator pattern and offers a developer-friendly access point to RDF repositories with methods to create and read RDF graphs. Various repository implementations may exist, for example database-specific implementations, or more generically, HTTP-based proxys or just simple SPARQL endpoints.

In order to compare the characteristics of SPARQL-based query plans and a native implementation over such a repository API, we assess validation performance by reasoning about the number of index lookups and scanned triples. We only use trivial property paths, i.e. predicate paths, and assume an indexing strategy that stores all six permutations of subject (S), predicate (P), and object (O).

We use a basic repository API that is similar to the interfaces commonly exposed by triplestores. We assume a function *getStatements* that receives three ordered arguments, subject, predicate, and object, for which the value *null* denotes a wildcard. The function returns an iterator over matching triples using one of the indices that starts with the parameters that do not have a wildcard as argument, e.g. if only subject receives a wildcard, then POS or OPS, if only predicate receives a

[1] `https://aws.amazon.com/neptune/`

wildcard, then SOP or OSP, or if both subject and object receives a wildcard, then POS or PSO. The iterator has a function *hasNext* which determines if there is a next triple and a function *next* which returns a structure with the properties *subject*, *predicate*, and *object* containing the next triple. We assume a function *registerViolationIf* that receives a predicate in first-order logic, and registers a violation if the predicate evaluates to true. We abstract away from any details, e.g. the violating node or cause of violation. Given the assumption that all six index permutations are available, this API provides us with an unbiased, perfect lookup for all triple patterns [36].

5.2 Query Plan Operators

A query plan in Amazon Neptune is a pipeline of operators. The first operator always is the *SolutionInjection* which injects static solutions. Each subsequent operator receives a set of incoming solutions and produces a new set of solutions that acts as input for the next operator in the pipeline. A solution can be understood as a relational table, whose column names are the variables introduced by preceding operators and whose values are possible assignments to these variables. The last operator, typically a *TermResolution* preceded by a *Projection*, defines the result of the query. We briefly explain some basic operators[2] in the Amazon Neptune query plans.

SolutionInjection Derives and injects static solutions from the query by combining various sources of static bindings (e.g., VALUES and BIND functions). If no solution can be derived, the universal solution is injected. All query plans start with this operator.

Distinct Computes the distinct projection on a subset of the variables specified by the *vars* argument, eliminating duplicates.

Filter Filters incoming solutions based on the filter condition defined by the *condition* argument.

Projection Projects over a subset of the variables specified by the *vars* argument. The operator has two modes that either retain or drop the specified variables.

PipelineJoin Joins each incoming solution against the tuple pattern defined by the *pattern* argument. The *joinProjectionVars* argument specifies the set of distinct projection variables when *distinct* is used. The *joinType* argument specifies one of the following join types to be performed: *join*, requires a join partner; *optional*, does not require a join partner; *minus*, requires that no join partner exists; and *existence check*, determines whether a join partner exists and binds the result to the *existenceCheckResultVar* variable.

HashIndexBuild Builds a hash index on the incoming solutions with as key the bindings for the variables specified by the *joinVars* argument and as value a list of the remaining bindings for each solution mapping to this key. When the *joinVars* argument is empty, all incoming solutions are mapped to the empty key. Outputs a distinct projection over the variables specified by the *joinVars* argument, i.e. the hash index keys. Subsequent operators can use this hash index solution set, i.e. its incoming solutions, referring it by its name specified by the *solutionSet* argument.

HashIndexJoin Joins the hash index solution set, referred to by the *solutionSet* argument, against the incoming solutions. The *joinType* and *existenceCheckResultVar* arguments have the same meaning as in the *PipelineJoin* operator.

TermResolution Translates between internal ID values and RDF terms of variables specified by the *vars* argument. The operator has two modes, *value2id* and *id2value*, that specify the mapping direction. Query plans typically end with this operator in *id2value* mode.

[2] https://docs.aws.amazon.com/neptune/latest/userguide/sparql-explain-operators.html

5.3 Constraint Components

EqualsConstraintComponent We start out with an example showing the query defined in the DASH namespace for validating constraints of kind *EqualsConstraintComponent*. We inject the static bindings used for generating its query plan.

```
1  SELECT DISTINCT ?this ?value
2  WHERE {
3    {
4      ?this ?PATH ?value .
5      MINUS {
6        ?this ?equals ?value .
7      }
8    }
9    UNION
10   {
11     ?this ?equals ?value .
12     MINUS {
13       ?this ?PATH ?value .
14     }
15   }
16   VALUES (?this ?PATH ?equals) {
17     (ex:v0 ex:P ex:Q)
18   }
19 }
```

The variables *this*, *PATH*, and *equals* are bound to the focus node, path property, and equals property, respectively. The goal is to extract subjects reachable from the focus node with either the value of the path or equals property as predicate, but not by both. This translates into to the union of terms reachable via the path but not equals property and terms reachable via the equals but not path property.

This is clearly visible in the query plan (Figure 5.1). The union on line 9 results in two similar branches where we focus on the left branch corresponding to line 3 to 8. The first *PipelineJoin* operator joins the initial solution against the pattern on line 4. This corresponds to extracting the subjects reachable via the path property. The *MINUS* function on line 5 results in a *HashIndexBuild* operator which builds a hash index on the variables shared between the incoming solutions and the pattern on line 6. The second *PipelineJoin* operator joins the hash index keys against the pattern defined on line 6. This corresponds to extracting the subjects reachable via the equals property that are also reachable via path property. Finally, the *HashIndexJoin* operator joins the hash index solution set against the incoming solutions, only outputting a hash index solution if no join partner exists. This corresponds to taking all subjects reachable via the path property, and subtracting a subset that is reachable via the equals (and path) property.

Having discussed the SPARQL-based query plan, we now turn towards a proposal for a native implementation:

```
result = getStatements($this, $PATH, null)

foreach (statement in result)
    subResult = getStatements($this, $equals, statement.object)

    registerViolationIf (subResult.hasNext)

result = getStatements($this, $equals, null)

foreach (statement in result)
    subResult = getStatements($this, $PATH, statement.object)

    registerViolationIf (subResult.hasNext)
```

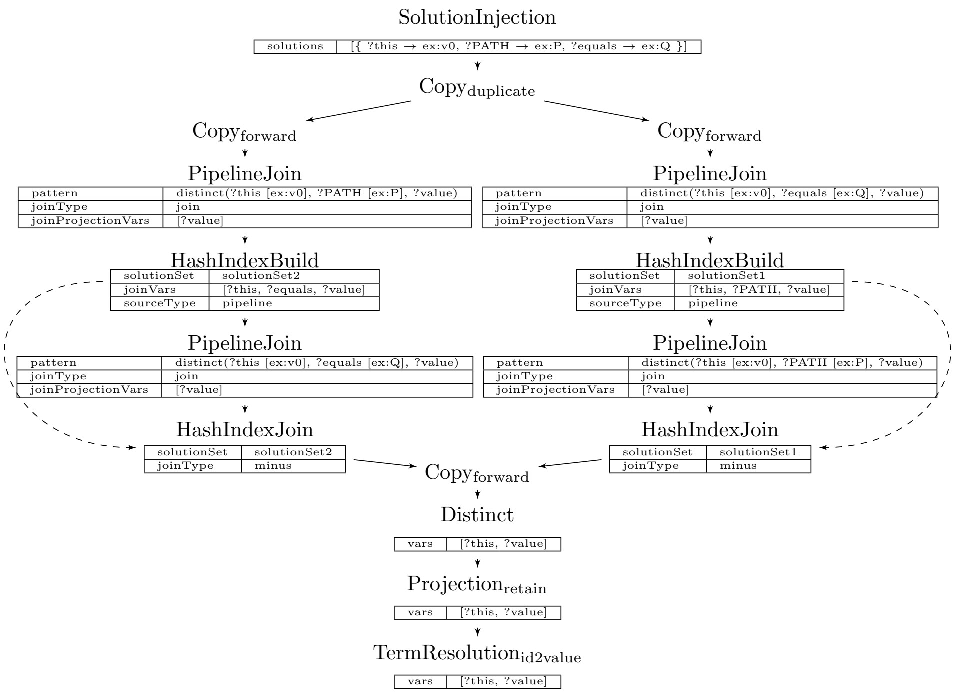


Figure 5.1: Query plan for *EqualsConstraintComponent*

Both the query plan and native implementation perform 2 SP lookups, scanning all n and m values, followed by exactly $n + m$ SPO lookups, each scanning at most 1 value.

HasValueConstraintComponent

```
1  SELECT ?this
2  WHERE {
3    FILTER NOT EXISTS {
4      ?this ?PATH ?hasValue
5    }
6    VALUES (this ?PATH ?hasValue) {
7      (ex:v0 ex:P ex:v1)
8    }
9  }
```

The variables *this*, *PATH*, and *hasValue* are bound to the focus node, path property, and value node, respectively. The goal is to extract the focus node if and only the value node is not reachable with the value of the path property as predicate.

This is clearly visible in the query plan (Figure 5.2). The *FILTER NOT EXISTS* function on line 3 results in a *HashIndexBuild* operator which builds a hash index on the variables shared between the initial solution and the pattern on line 6. Trivially, this results in a hash index of one key. The *PipelineJoin* operator joins the hash index key against the pattern defined on line 6. This corresponds to extracting the value node reachable via the path property. The *HashIndexJoin* operator joins the hash index solution against the incoming solution, adding a variable indicating whether a join partner exists, i.e. whether the value node is reachable. Finally, the *Filter* operator filters the incoming solution based on this variable.

Native implementation:

```
result = getStatements($this, $PATH, $hasValue)

registerViolationIf (result.hasNext)
```

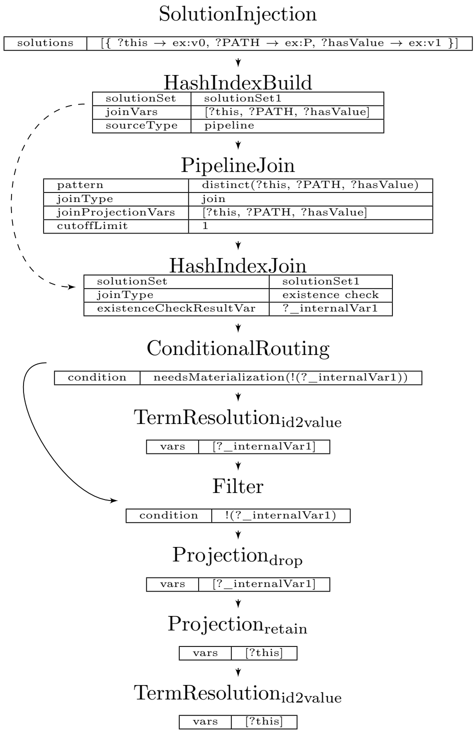

SolutionInjection
solutions | [{ ?this → ex:v0, ?PATH → ex:P, ?hasValue → ex:v1 }]
HashIndexBuild
solutionSet | solutionSet1
joinVars | [?this, ?PATH, ?hasValue]
sourceType | pipeline
PipelineJoin
pattern | distinct(?this, ?PATH, ?hasValue)
joinType | join
joinProjectionVars | [?this, ?PATH, ?hasValue]
cutoffLimit | 1
HashIndexJoin
solutionSet | solutionSet1
joinType | existence check
existenceCheckResultVar | ?_internalVar1
ConditionalRouting
condition | needsMaterialization(!(?_internalVar1))
TermResolution_id2value
vars | [?_internalVar1]
Filter
condition | !(?_internalVar1)
Projection_drop
vars | [?_internalVar1]
Projection_retain
vars | [?this]
TermResolution_id2value
vars | [?this]

Figure 5.2: Query plan for *HasValueConstraintComponent*

Both the query plan and native implementation perform 1 SPO lookup, scanning at most 1 value.

LessThanConstraintComponent

```
1  SELECT ?this ?value
2  WHERE {
3    ?this ?PATH ?value .
4    ?this ?lessThan ?otherValue .
5    BIND (?value < ?otherValue AS ?result) .
6    FILTER (!bound(?result) || !(?result)) .
7    VALUES (?this ?PATH ?lessThan) {
8      (ex:v0 ex:P ex:Q)
9    }
10 }
```

The variables *this*, *PATH*, and *lessThan* are bound to the focus node, path property, and less than property, respectively. The goal is to extract subjects reachable from the focus node with the value of the path property as predicate for which a subject exists that is reachable with the value of less than property as predicate and incomparable or not arithmetically greater.

This is clearly visible in the query plan (Figure 5.7). The first *PipelineJoin* operator joins the initial solution against the pattern on line 3. This corresponds to extracting the subjects reachable via the path property. The second *PipelineJoin* operator joins these against the pattern on line 4. This corresponds to extracting the cartesian product of subjects reachable via the path property and less than property. The *BIND* function on line 5 results in a *Filter* operator. This corresponds to comparing the subjects and binding the result to a variable. When incomparable, the variables remains unbound. Finally, the *FILTER* function on line 6 results in a *Filter* operator, filtering the incoming solutions based on this variable.

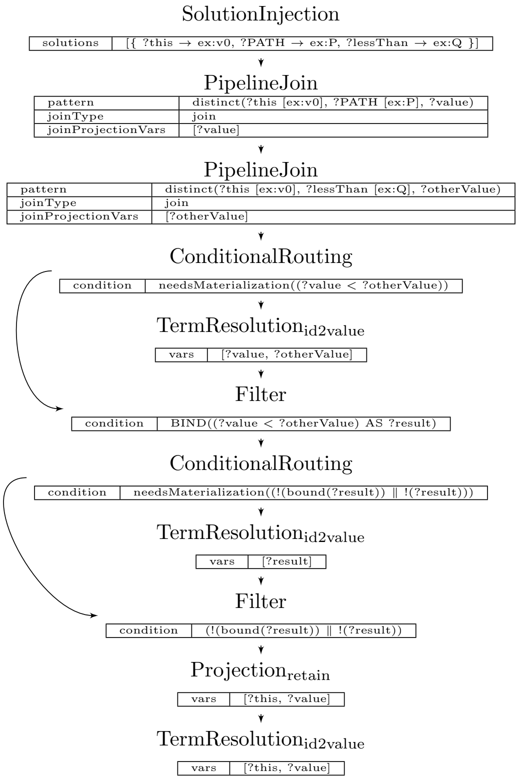


Figure 5.3: Query plan for *LessThanConstraintComponent*

Native implementation:

```
result1 = getStatements($this, $PATH, null)
result2 = getStatements($this, $lessThan, null)

foreach (statement1 in result1)
    foreach (statement2 in result2)
        registerViolationIf(statement1.object < result2.object)
```

The query plan performs 1 SP lookup, scanning all n values, followed by exactly n SP lookups, each scanning all m values. The native implementation performs 2 SP lookups but still scans nm values.

The native complexity can be reduced from quadratic to linear by returning a non-compliant validation report. When one is not interested in the exact value causing the failure, the inner loop can be moved outside the outer loop. Hereby reducing the number of scanned values from nm to $n + m$.

Native implementation:

```
result = getStatements($this, $lessThan, null)
minimum = null

foreach(statement in result)
    if (minimum == null)
        minimum = statement.object
    else
        minimum = min(minimum, statement.object)

if (minimum != null)
    result = getStatements($this, $PATH, null)

    foreach (statement in result)
       registerViolationIf (statement.object < minimum)
```

LessThanOrEqualsConstraintComponent The query, query plan, and native implementation are identical to those of *LessThanConstraintComponent* but with operator >= instead of >.

MaxCountConstraintComponent

```
1 SELECT ?this
2 WHERE {
3   ?this ?PATH ?value .
4   VALUES (?this ?PATH ?maxCount) {
5     (ex:v0 ex:P 1)
6   }
7 }
8 GROUP BY ?this
9 HAVING (COUNT(DISTINCT ?value) > ?maxCount)
```

The variables *this*, *PATH*, and *maxCount* are bound to the focus node, path property, and maximum number, respectively. The goal is to extract the focus node if and only if the number of subjects reachable with the value of the path property as predicate is greater than the maximum number.

This is clearly visible in the query plan (Figure 5.4). The *PipelineJoin* operator joins the initial solution against the pattern on line 3. This corresponds to extracting the subjects reachable via the path property. The *GROUP BY* and *HAVING* functions on line 8 and 9 result in a *Aggregate* operator, outputting a solution if and only if the number of distinct subjects is greater than the maximum number. Unintentionally, the *maxCount* variable remains unbound in the *having* argument as it is not part of the *groupBy* argument. The semantics of Apache Jena differ with Amazon Neptune and Blazegraph. The intended semantics can be acquired by substituting the variable with the maximum number or by adding the variable to the *groupBy* argument.

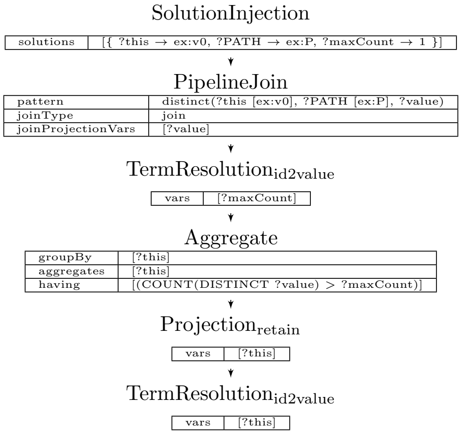


Figure 5.4: Query plan for *MaxCountConstraintComponent*

Native implementation:

```
result = getStatements($this, $PATH, null)
count = 0

while (result.next and count < $maxCount + 1)
    count = count + 1

registerViolationIf (count <= $maxCount)
```

The query plan performs 1 SP lookup, scanning all n values. The native implementation performs 1 SP lookup, scanning $\min(n, maxCount + 1)$ values.

MinCountConstraintComponent

```
1  SELECT ?this
2  WHERE {
3    OPTIONAL {
4      ?this ?PATH ?value .
5    }
6    VALUES (?this ?PATH ?minCount) {
7      (ex:v0 ex:P 1)
8    }
9  }
10 GROUP BY ?this
11 HAVING (COUNT(DISTINCT ?value) < ?minCount)
```

The variables *this*, *PATH*, and *minCount* are bound to the focus node, path property, and minimum number, respectively. The goal is to extract the focus node if and only if the number of subjects reachable with the value of the path property as predicate is smaller than the minimum number.

This is clearly visible in the query plan (Figure 5.5). The *PipelineJoin* operator joins the initial solution against the pattern on line 4. This corresponds to extracting the subjects reachable via the path property. The *GROUP BY* and *HAVING* functions on line 10 and 11 result in a *Aggregate* operator, outputting a solution if and only if the number of distinct subjects is smaller than the smaller number. Unintentionally, the *minCount* variable remains unbound in the *having* argument as it is not part of the *groupBy* argument. The semantics of Apache Jena differ with Amazon Neptune and Blazegraph. The intended semantics can be acquired by substituting the variable with the minimum number or by adding the variable to the *groupBy* argument.

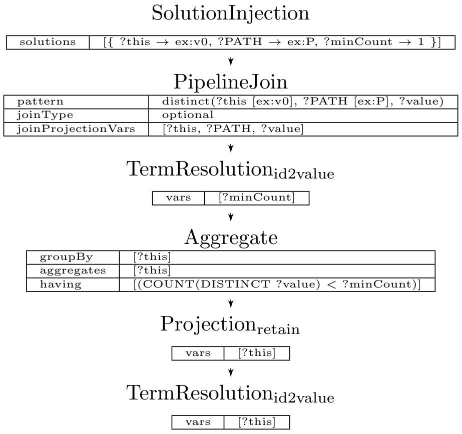

SolutionInjection
solutions
[{ ?this → ex:v0, ?PATH → ex:P, ?minCount → 1 }]
PipelineJoin
pattern
distinct(?this [ex:v0], ?PATH [ex:P], ?value)
joinType
optional
joinProjectionVars
[?this, ?PATH, ?value]
TermResolution$_{\text{id2value}}$
vars
[?minCount]
Aggregate
groupBy
[?this]
aggregates
[?this]
having
[(COUNT(DISTINCT ?value) < ?minCount)]
Projection$_{\text{retain}}$
vars
[?this]
TermResolution$_{\text{id2value}}$
vars
[?this]

Figure 5.5: Query plan for *MinCountConstraintComponent*

Native implementation:

```
result = getStatements($this, $PATH, null)
count = 0

while (result.next && count < $minCount)
    count = count + 1

registerViolationIf (count == $minCount)
```

The query plan performs 1 SP lookup, scanning all n values. The native implementation performs 1 SP lookup, scanning $\min(n, minCount)$ values.

DisjointConstraintComponent

```
ASK {
  FILTER NOT EXISTS {
    ?this ?disjoint ?value .
  }
}
```

ASK queries simply return a boolean indicating whether a solution exists. In order to find and report the value nodes causing a violation, ASK queries must be transformed into SELECT queries. We use the procedure implemented in TopBraid's SHACL API. TopBraid's SHACL API is a SHACL validator based on the DASH namespace and will be discussed in detail in Section 6.1. The query is transformed into the following query:

```
1  SELECT DISTINCT ?this ?value
2  WHERE {
3    ?this ?PATH ?value .
4    FILTER NOT EXISTS {
5      FILTER NOT EXISTS {
6        ?this ?disjoint ?value .
7      }
8    }
9    VALUES (?this ?PATH ?disjoint) {
10     (ex:v0 ex:P ex:Q)
11   }
12 }
```

The variables *this*, *PATH*, and *disjoint* are bound to the focus node, path property, and disjoint property, respectively. The goal is to extract subjects reachable from the focus node with the value of the path property as predicate that are also reachable with the value of the disjoint property as predicate.

This query results in a query plan of 18 stages (Figure 5.6). The *FILTER NOT EXISTS* function on line 4 and 5 result in two *HashIndexBuild* operators that build a hash index on the variables shared between their inner and outer scopes. There are no shared viables as the variables on line 3 are not used in the first *FILTER NOT EXISTS* function, Therefore they do not carry over to the second function. As result the *joinVars* argument remains empty. When the join variables differ from the projection variables, the variables needed inside, i.e. *disjoint* and *this*, the operator shows alternative semantics. In this case the operator does not output the distinct projection over the join variables, but the incoming solutions instead, followed by a *Distinct* operator. In the second *FILTER NOT EXISTS* function, both the join and projection variables empty, hence it outputs the empty projection over the join variables, i.e. the empty solution. All variables in the pattern of the next *PipelineJoin* operator incorrectly remain unbound. The semantics in Amazon Neptune and Blazegraph are incorrect as results differ from the results obtained by evaluating in the SPARQL algebra.

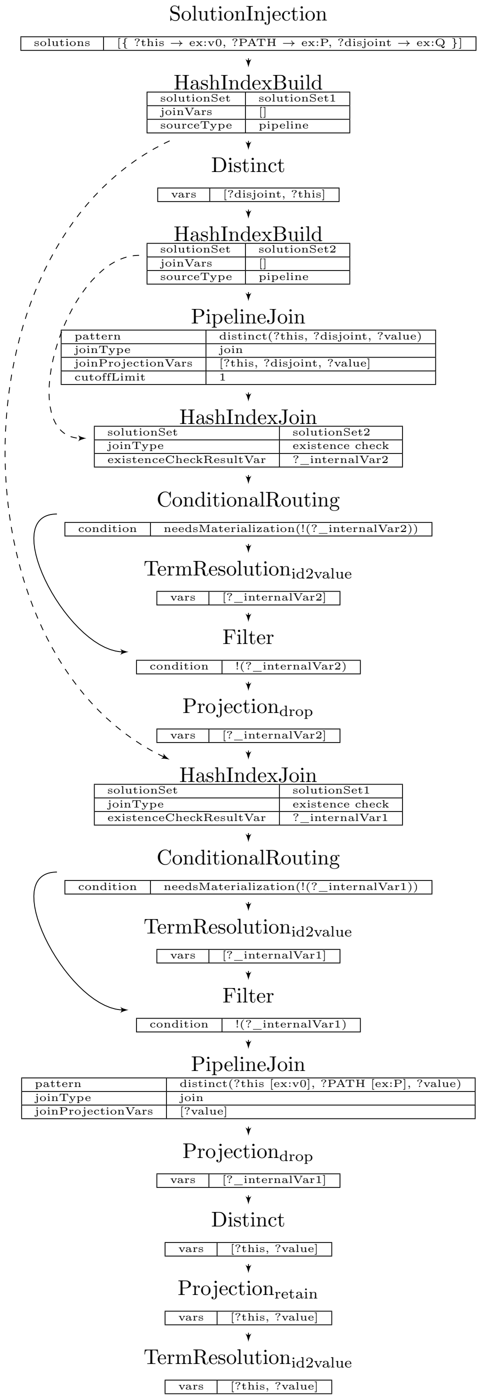

SolutionInjection
solutions | [{ ?this → ex:v0, ?PATH → ex:P, ?disjoint → ex:Q }]
HashIndexBuild
solutionSet | solutionSet1
joinVars | []
sourceType | pipeline
Distinct
vars | [?disjoint, ?this]
HashIndexBuild
solutionSet | solutionSet2
joinVars | []
sourceType | pipeline
PipelineJoin
pattern | distinct(?this, ?disjoint, ?value)
joinType | join
joinProjectionVars | [?this, ?disjoint, ?value]
cutoffLimit | 1
HashIndexJoin
solutionSet | solutionSet2
joinType | existence check
existenceCheckResultVar | ?_internalVar2
ConditionalRouting
condition | needsMaterialization(!(?_internalVar2))
TermResolution$_{\text{id2value}}$
vars | [?_internalVar2]
Filter
condition | !(?_internalVar2)
Projection$_{\text{drop}}$
vars | [?_internalVar2]
HashIndexJoin
solutionSet | solutionSet1
joinType | existence check
existenceCheckResultVar | ?_internalVar1
ConditionalRouting
condition | needsMaterialization(!(?_internalVar1))
TermResolution$_{\text{id2value}}$
vars | [?_internalVar1]
Filter
condition | !(?_internalVar1)
PipelineJoin
pattern | distinct(?this [ex:v0], ?PATH [ex:P], ?value)
joinType | join
joinProjectionVars | [?value]
Projection$_{\text{drop}}$
vars | [?_internalVar1]
Distinct
vars | [?this, ?value]
Projection$_{\text{retain}}$
vars | [?this, ?value]
TermResolution$_{\text{id2value}}$
vars | [?this, ?value]

Figure 5.6: Query plan 1 for *DisjointConstraintComponent*

The query above can actually be simplified to a join between the subjects reachable via the path property against its counterpart, these subjects reachable via the disjoint property:

```
1 SELECT DISTINCT ?this ?value
2 WHERE {
3   ?this ?PATH ?value .
4   ?this ?disjoint ?value .
5   VALUES (?this ?PATH ?disjoint) {
6     (ex:v0 ex:P ex:Q)
7   }
8 }
```

The variables *this*, *PATH*, and *disjoint* are bound to the focus node, path property, and disjoint property, respectively.

This query results in a query plan of 6 stages (Figure 5.7) that clearly corresponds to the intended semantics. The first *PipelineJoin* operator joins the initial solution against the pattern on line 3. This corresponds to extracting the subjects reachable via the path property. The second *PipelineJoin* operator joins these against the pattern on line 4. This corresponds to extracting the subjects reachable via the path property that are also reachable by the disjoint property.

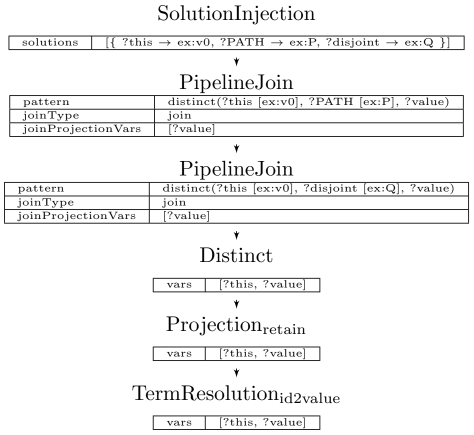


Figure 5.7: Query plan 2 for *DisjointConstraintComponent*

Native implementation:

```
result = getStatements($this, $PATH, null)

foreach (statement in result)
    subResult = getStatements($this, $disjoint, statement.object)

    registerViolationIf (!subResult.hasNext)
```

Both the query plan and native implementation perform 1 SP lookup, scanning all n values, followed by exactly n SPO lookups, each scanning at most 1 value.

5.4 Conclusions

For the constraint components *EqualsConstraintComponent*, *HasValueConstraintComponent*, and *DisjointConstraintComponent* we found no difference between native and SPARQL-based validation using Amazon Neptune in terms of the number of index lookups and scanned triples. The *LessThanConstraintComponent* and *LessThanOrEqualsConstraintComponent* have a constant number of index lookups in the native implementation compared to a linear number for SPARQL-based validation. Both however, scan the same number of triples in total.

A native implementation allows control over the iterator which can be used to limit the number of scanned triples (e.g., early termination). This is done for *MinCountConstraintComponent* and *MaxCountConstraintComponent* when it is certain that the constraint has been satisfied or violated, i.e. when the minimum or maximum cardinality has been exceeded, respectively. SPARQL-based validation using Amazon Neptune scans and then aggregates all the triples. The scan operator in Amazon Neptune, *PipelineJoin*, has an argument *cutoffLimit* that can be used to limit the number of extracted join partners[3], i.e. the number of scanned triples. The query plan generator can be modified to recognize minimum and maximum cardinality constraint patterns, and for these cases apply an appropriate *cutoffLimit* to achieve the same advantage that the native implementation currently has over SPARQL-based validation.

In general, a native implementation is more likely to use less memory compared to SPARQL-based validation as it does not use large intermediate result sets that could significantly increase in size due to join operations. The constraints we have looked at are relatively simple and not complex, and may therefore benefit from avoiding the overhead induced by parsing and optimizing SPARQL queries.

[3] `https://docs.aws.amazon.com/neptune/latest/userguide/sparql-explain-operators.html`

Chapter 6

A Practical Algorithm for Validating SHACL

6.1 TopQuadrant's SHACL API

TopQuadrant maintained DASH [21], developed SPIN [22], and played an important role in the development of SHACL [23]. TopQuadrant's SHACL API[1] is based on Apache Jena[2] and the DASH validators. Apache Jena is an open-source Java framework for building Semantic Web and Linked Data applications. It offers an RDF API to interact with the core API to create and read RDF graphs, a SPARQL engine called ARQ for querying, triple stores, and an ontology and inference API.

Algorithm Validation of a data graph against a shapes graph in TopQuadrant's SHACL API can be described as follows:

1. (Inference) The first step prior to validation is an inferencing step. The goal is to produce a data graph containing all triples inferred by the entailment regime specified by the *sh:entailment* property. The inferencing is performed by Apache Jena.
2. (Construction of root shapes) Then a set of non-deactivated root shapes is constructed. These are subjects of triples that have *sh:targetClass*, *sh:targetNode*, *sh:targetObjectsOf* or *sh:targetSubjectsOf* as predicate.
3. For each root shape s:
   (a) (Construction of constraints) The set of constraints is constructed by iterating over a shape's properties and then looking up, for each property, the related constraint component (having the property as parameter) in the SHACL namespace. If the constraint component has a single parameter, then each value for the property becomes a constraint, else the constraint is constructed only when the shape declares values for all mandatory parameters.
   (b) (Execution of constraint executors) Lastly, the tool validates each constraint against the shape's focus nodes by means of a dedicated constraint component executor.

Constraint Executors Constraint executors execute the validation of a set of focus nodes against a constraint. TopQuadrant's SHACL API has dedicated constraint executors for the constraint components *PropertyConstraintComponent*, *JsConstraintComponent*, *SparqlConstraintComponent*, and *ExpressionConstraintComponent*. Other constraint components are handled by

[1] `https://github.com/TopQuadrant/shacl`

[2] `https://jena.apache.org/`

the *SPARQLComponentExecutor* or *JSComponentExecutor* executors that operate by means of the validators defined in the DASH namespace. The latest version of TopQuadrant's SHACL, starting from version 1.3.0, provides dedicated constraint executors for all SHACL constraint components.

The *SPARQLComponentExecutor* executor validates constraints, using the SPARQL queries discussed in detail in the previous chapter. For a constraint c of kind C, the tool looks for triples of the form $(C, p, ?v)$ where p is *sh:propertyValidator* if c is declared by a property shape or *sh:nodeValidator* if c is declared by a node shape. Based on whether v is an instance of class *sh:SPARQLSelectValidator* or *sh:SPARQLAskValidator* the value of property *sh:select* or *sh:ask* is used as SELECT or ASK query, respectively. ASK queries are transformed into SELECT queries according to the following pattern:

```
SELECT DISTINCT ?this ?value
WHERE {
  ?this ?PATH ?value
  FILTER NOT EXISTS { %QUERY PATTERN% }
}
```

for which the query pattern of the ASK query is substituted.

Each query binds the variable *this* to the focus node, as well as any variables that represent the parameters of the constraint component. Constraints declared by property shapes bind or substitute the *PATH* variable according to the SHACL property path, i.e. the value of *sh:path* for the shape. If the value is an IRI, i.e. predicate path, then the value can be bound, else the value is a blank node, i.e. sequence, alternative, inverse, zero-or-more, one-or-more, or zero-or-one path, and the SPARQL equivalent is substituted.

These queries retrieve the witnesses violating the constraint for a specific focus node. If a solution exists, then the focus node is not valid against the constraint.

References DASH does not support references or nested shapes, which implies that Example 27 cannot be validated solely using DASH. TopQuadrant's SHACL API solves this limitation by registering a custom SPARQL Value Function. Such functions are registered in ARQ's function registry (a mapping from URI to a factory class for functions) and executed during query evaluation.

The TopBraid Data Shapes Library (TOSH) namespace defines a validator for the *NodeConstraintComponent*. The SPARQL query uses the registered *tosh:hasShape* function which validates a focus node against a shape. This allows for references as demonstrated by Example 32.

Recursion TopQuadrant's SHACL API does not support recursion. It implements a recursion guard to avoid infinite recursion. More precisely, it avoids recursive calls to the *tosh:hashShape* function (used to handle references) by keeping track of the focus nodes and shapes currently being recursed.

Example 32. We validate the following data and shapes graph, requiring that everything liked by James has a nationality and that he likes at least one such thing:

```
ex:JamesShape
  a sh:NodeShape ;
  sh:targetNode ex:James ;
  sh:property ex:LikesShape .

ex:LikesShape
  a sh:PropertyShape ;
  sh:path ex:likes ;
  sh:minCount 1 ;
  sh:node ex:ThingShape .

ex:ThingShape
  a sh:NodeShape ;
  sh:property ex:NationalityShape .

ex:NationalityShape
  a sh:PropertyShape ;
  sh:path ex:nationality ;
  sh:minCount 1 .
```

Shapes graph

```
ex:James ex:likes ex:Mary .
ex:Mary ex:nationality "British" .
```

Data graph

The set of root shapes only consists of the shape *ex:JamesShape* as the other shapes do not declare any target. The root shape *ex:JamesShapes* has the property *sh:property* which is a parameter of constraint component *PropertyConstraintComponent*. The validator validates each focus node in the target of the shape against the validators of constraints declared by the shape. This results in the validation of focus node *ex:James* against the validator of *PropertyConstraintComponent*. The validator of *PropertyConstraintComponent* validates each value node against the given property shape, i.e. the value of *sh:property*. This results in the validation of focus node *ex:James* against the shape *ex:LikesShape*.

The shape *ex:LikesShape* has the properties *sh:minCount* and *sh:node* which are parameters of constraint components *MinCountConstraintComponent* and *NodeCountConstraintComponent*, respectively. The validator validates each focus node against the constraints declared by the shape. This results in the validation of focus node *ex:James* against a constraint of kind *MinCountConstraintComponent* and *NodeCountConstraintComponent*.

The validator of *MinCountConstraintComponent* uses the following SPARQL query (Section 5.3):

```
SELECT ?this
WHERE {
  OPTIONAL {
    ?this ?PATH ?value .
  }
  VALUES (?this ?PATH ?minCount) {
    (ex:James ex:likes 1)
  }
}
GROUP BY ?this
HAVING (COUNT(DISTINCT ?value) < ?minCount)
```

which requires at least one value node to exist. The focus node *ex:James* conforms to this constraint as it has *ex:Mary* as value for *sh:likes*, i.e. there is no solution to the query.

The validator of *NodeConstraintComponent* uses the following SPARQL query (Section 5.3):

```
SELECT DISTINCT ?this ?value ?failure
WHERE {
  ?this ?PATH ?value
  BIND(tosh:hasShape(?value, ?node) AS ?hasShape)
  BIND(( ! bound(?hasShape) ) AS ?failure)
  FILTER ( ?failure || ( ! ?hasShape ) )
  VALUES (?this ?PATH ?node) {
    (ex:James ex:likes ex:ThingShape)
  }
}
```

which requires all value nodes to conform to *ex:ThingShape*. Evaluation of the query invokes the *tosh:hasShape* function registered in the ARQ function registry. It receives the parameter value *ex:ThingShape* and a single value node *ex:Mary*. This results in the validation of value node *ex:Mary* as focus node against the shape *ex:ThingShape*.

The shape *ex:ThingShape* has the property *sh:property* which is a parameter of constraint component *PropertyConstraintComponent*. This results in the validation of focus node *ex:Mary* against the shape *ex:NationalityShape*. The shape *ex:NationalityShape* has the property *sh:minCount* which is a parameter of constraint component *MinCountConstraintComponent*. The focus node *ex:Mary* conforms to this constraint as it has "British" as value for *sh:nationality*. We conclude that focus node *ex:Mary* conforms to the shape *ex:NationalityShape* as it conforms to the minimum cardinality constraint. Then value node *ex:Mary* conforms to the shape *ex:ThingShape* as it conforms to the property constraint.

We conclude that focus node *ex:James* conforms to the constraint of kind *NodeConstraintComponent*, i.e. there is no solution to the query. Then value value node *ex:James* conforms to the shape *ex:LikesShape* as it conforms to both declared constraints. We conclude that focus node *ex:James* conforms to the shape *ex:JamesShape* as it conforms to the property constraint. So the data graph is valid with respect to the shapes graph. △

6.2 Recursive SHACL API

6.2.1 Introduction

Immediate validation of referencing constraints (see Section 4.3) leads to nested validation of value nodes against the referenced shapes. If the constraint-declaring shape is a property shapes then the value nodes are the nodes reachable from the focus nodes by the property path. Immediate validation of these constraints may lead to infinite recursion when the declaring shape is recursive and the value node part of a cycle in the data graph.

We propose a new native algorithm in Section 6.2.2 for validating non-recursive SHACL using immediate constraint validation. The algorithm is based on SHACL's definitions of validation [23].

We also propose a new native hybrid algorithm in Section 6.2.3 to validate recursive SHACL and mitigate the infinite recursion problem by extending the algorithm for non-recursive SHACL with a minimal fixed-point algorithm handling validation against recursive shapes. The fixed-point algorithm produces a minimal fixed-point assignment σ_{minFix} as defined in Section 4.4. The validation against recursive shapes is based on Theorem 2, which states that a node v_0 is valid against a shape s if and only if $\neg s \notin \sigma_{\text{minFix}}(v_0)$. The hybrid algorithm is sound and complete for all tractable fragments, including the non-recursive SHACL fragment.

We improve the original fixed-point algorithm [10] by:

1. avoiding redundant validations using monotonicity and constraint deactivation,

2. minimizing the number of shapes to be assigned using the reference closure of recursive shapes,

3. minimizing the number of nodes getting shapes assigned by calculating all value nodes for each shape,
4. minimizing the number of inconclusive answers by adding an explicit ordering to the validation against shapes, and
5. adding the ability to generate SHACL-like validation reports indicating the cause of violations.

Notation The variable R is the set of validation results, G_S a shapes graph, G_D a data graph, and G^V the nodes of graph G.

Abstractions We abstract away from failures, i.e. the handling of exceptions, and from the deactivation of shapes that exclude a shape from the validation context. We assume a function $getValueNodes$ that receives a data graph, shapes graph, shape, and focus node, and returns the value nodes for the given focus node and any constraint declared by the given shape.

6.2.2 Algorithm for Immediate Constraint Evaluation

Validation is a mapping from an input (i.e., data graph and shapes graph, data graph and shape, focus node and shape, or focus node and constraint) to validations results. The validation results of a data graph against a shapes graph is the union of validation results of the data graph against all shapes in the shapes graph. The validation results of a data graph against a shape are the union of validation results of all focus nodes that are in the target of the shape. The validation results of a focus node against a shape is the union of validation results of the focus node against all constraints declared by the shape. Unless the shape has been deactivated, in that case the validation results are empty. [23]

Since the validation results against any constraint declared by a deactivated shape is empty, and because nothing besides the validation against a shape triggers the validation against a constraint, we can move up the condition to produce empty validation results if the shape has been deactivated to the validation against a shape.

Similarly for the validation of a data graph. The validation of a data graph against a shape consists of the validation of all focus nodes that are in the target of the shape. The validation results may only be non-empty if the shape declares a target and is non-deactivated. So we can redefine the validation of a data graph against a shapes graph as follows:

Definition 6.2.1. (Validation of a data graph against a shapes graph) The validation results of a data graph against a shapes graph is the union of validation results of the data graph against all target-declaring non-deactivated shapes in the shapes graph.

We assume a function *getRootShapes* that receives a shapes graph and returns the set of target-declaring non-deactivated shapes in the given shapes graph. The validation of a data graph against a shapes graph (Definition 6.2.1) is given by Algorithm 1.

Algorithm 1 Validation of a data graph against a shapes graph

```
procedure validateDataGraphAgainstShapesGraph($G_D, G_S$)
    $R \leftarrow \emptyset$
    for all $s \in$ getRootShapes($G_S$) do
        $R \leftarrow R \cup$ validateDataGraphAgainstShape($G_D, G_S, s$)
    end for
    return $R$
end procedure
```

We redefine the validation of a data graph against a shape and the validation of a focus node against a shape to consider a set of focus nodes instead of a single focus node. We now avoid invoking the algorithm for validating against a shape for each focus node, instead it is being invoked once for a set of focus nodes, hereby limiting the call stack and overhead induced by initialization (e.g., retrieving the constraints declared by the shape).

Definition 6.2.2. (Validation of a data graph against a shape) The validation results of a data graph against a shape are the validation results of the set of focus nodes that are in the target of the shape.

Definition 6.2.3. (Validation of a set of focus nodes against a shape) The validation results of a set of focus nodes against a shape is the union of validation results of the set of focus nodes against all constraints declared by the shape.

We assume a function *getTarget* that receives a data graph, shapes graph, and shape, and returns the focus nodes in the given data graph identified by one or more target declarations of the given shape in the given shapes graph. The validation of a data graph against a shape (Definition 6.2.2) is given by Algorithm 2.

Algorithm 2 Validation of a data graph against a shape

```
procedure validateDataGraphAgainstShape($G_D, G_S, s$)
    $V_{\text{focus}} \leftarrow$ getTarget($G_D, G_S, s$)
    if $|V_{\text{focus}}| > 0$ then
        return validateNodesAgainstShape($G_D, G_S, V_{\text{focus}}, s$)
    end if
    return $\emptyset$
end procedure
```

We further assume a function *getConstraints* that receives a shapes graph and shape, and returns all constraints declared by the given shape in the given shapes graph. We represent constraints by a tuple (s, C, P) where s is the constraint-declaring shape, C the constraint component IRI, and P a mapping from mandatory parameters to their values. The validation of a set of focus nodes against a shape (Definition 6.2.3) is given by Algorithm 3.

Algorithm 3 Validation of a set of focus nodes against a shape

```
procedure validateNodesAgainstShape($G_D, G_S, V_{\text{focus}}, s$)
    $R \leftarrow \emptyset$
    for all $c \in$ getConstraints($G_S, s$) do
        $R \leftarrow R \cup$ validateNodesAgainstConstraint($G_D, G_S, V_{\text{focus}}, c$)
    end for
    return $R$
end procedure
```

A focus node conforms to a shape if and only if the validation results of the focus node against the shape is empty and no failure has been reported by it [23]. Most referencing constraint components rely on conformance checking. The validation results used to determine the outcome of conformance checking are separated from the surrounding validation process.

The validation results of a focus node against a constraint of kind C is defined by the validator of the constraint component C [23]. We redefine the validation of a focus node against a constraint, for similar reasons as before, to consider a set of focus nodes instead of a single focus node:

Definition 6.2.4. (Validation of a set of focus nodes against a constraint) The validation results of a set of focus nodes against a constraint of kind C is defined by the validator of the constraint component C.

We assume a function F that maps a constraint component IRI to its validator. A validator of constraint component C receives a data graph, shapes graph, set of focus nodes, and constraint of kind C, and returns the validation results of the set of focus nodes against the constraint of kind C. The validation of a set of focus nodes against a constraint (Definition 6.2.4) is given by Algorithm 4.

Algorithm 4 Validation of a set of focus nodes against a constraint

```
procedure validateNodesAgainstConstraint($G_D, G_S, V_{\text{focus}}, c$)
    $(s, C, P) \leftarrow c$
    return $\mathrm{F}(C)(G_D, G_S, V_{\text{focus}}, c)$
end procedure
```

Figure 6.2 visualizes the interaction between algorithms, i.e. how Algorithms 1 to 4 are related.

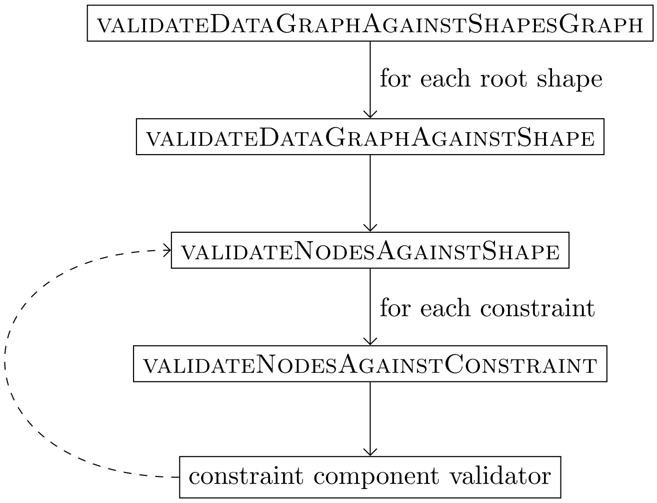

validateDataGraphAgainstShapesGraph
for each root shape
validateDataGraphAgainstShape
validateNodesAgainstShape
for each constraint
validateNodesAgainstConstraint
constraint component validator

Figure 6.2: Interaction between algorithms

Constraint Components The validator of a constraint component C validates a set of focus nodes against constraints of kind C. Value nodes for each focus node are retrieved and validated with regards to their adherence to the requirements specified by the constraint.

The validation results of a set of focus nodes against referencing constraint components requiring validation (i.e., *PropertyConstraintComponent*) is defined as the nested validation of the set of value nodes as focus nodes against the referenced shapes. The validation against referencing constraints components requiring conformance checking (e.g., *NodeConstraintComponent*) produces validation results based on whether value nodes conform to the referenced shapes, i.e. whether the nested validation results are empty. Lastly, the validation results against non-referencing constraint components (e.g., *EqualsConstraintComponent*) do not result in nested validation, but instead, is solely based on the value nodes.

Appendix 1.2 describes and provides algorithms for the validation of one constraint component of each type for non-recursive SHACL.

6.2.3 Algorithm for Recursive SHACL

We extend the algorithm for non-recursive SHACL, introduced in previous section, with a minimal fixed-point algorithm handling validation against recursive shapes. The validation against recursive shapes consists of two steps. In the first step a fixed-point algorithms attempts to assign shapes to the focus nodes and their value nodes. The shapes to be assigned and the nodes getting these shapes assigned are referred to as fixed-point shapes and nodes, respectively. In the second step the validation results of the non-conforming focus nodes against the shape are generated using the minimal fixed-point assignment of the previous step.

In contrast to immediate constraint evaluation, validation against referencing constraints does not result in nested validation against the referenced shapes, but instead, uses the assignment. This is similar to the inductive evaluation of a constraint formula presented in Section 4.4.3. The 3-valued logic, i.e. the introduction of an unknown truth value, prevents deciding conformance by simply checking if validation results are empty. Instead, one needs to be able to distinguish between all three values. Therefore, we informally introduce a dedicated vocabulary with namespace IRI *rsh:* for referencing constraints to propagate conformance to the surrounding validation process. The vocabulary has the properties *rsh:true*, *rsh:false*, and *rsh:unknown* to declare the conformance of focus nodes to the constraint-declaring shape.

We extend the validation against a constraint with two additional parameters:

- an optional assignment σ whose presence indicates that referencing constraints should not perform nested validation, but use the assignment instead; and
- an optional boolean flag b that indicates whether referencing constraints must produce conformance results, i.e. when the validation is invoked by the fixed-point algorithm.

Improvements The original algorithm [10] attempts to assign all shapes to all nodes in the graph, regardless of their targets. We minimize the set of fixed-point shapes as we only use the fixed-point algorithm for the validation against recursive shapes, and the set of fixed-point nodes by calculating their value nodes:

- Fixed-point shapes: Only shapes needed to decide the conformance against the recursive shape need to be assigned, i.e. all shapes reachable from the recursive shape in the dependency graph, the reference closure. The reference closure of a recursive shape can be determined using standard DFS with cycle detection on the dependency graph.
- Fixed-point nodes: A fixed-point shape only needs to be assigned to nodes that become value nodes in the validation against referencing constraints referencing this fixed-point shape. These nodes can be determined by a modified version of DFS with cycle detection that considers node-shape pairs instead of just a node. The idea is to explore the data graph starting from each focus node with the recursive shape as shape, keep track of explored node-shape pairs, and when a shape has references, i.e. declares referencing constraints, explore each value node for each referenced shape.

  The function *getFixedPointNodes* builds such a mapping from (fixed-point) shape to a set of fixed-point nodes and is given by Algorithm 5, while abusing notation by writing $v(s) \in \zeta$ instead of $v \in \zeta(s)$ (similarly for $\neg s$).

We limit inconclusive answers, i.e. an unknown truth value, by processing referenced shapes before the shapes that reference them. Such a topological ordering can be determined by using a modified version of DFS with cycle detection on the dependency graph that adds the node to a stack after having explored its adjacent nodes. This can be done in the same pass in which the reference closure of a recursive shape is determined.

The effective set of fixed-point nodes for a fixed-point shape is non-strictly decreasing as the assignment is preserved over consecutive iterations due to monotonicity. Therefore, fixed-point shapes only need to be assigned to fixed-point nodes that do not have the shape or its negation assigned.

We deactivate non-referencing constraints after the first iteration as the validation results of the set of effective fixed-point nodes against non-referencing constraints in subsequent iterations can only be empty. The validation of non-referencing constraints does not depend on the assignment. So after the first iteration, any effective fixed-point node must be valid against non-referencing constraints, else a violation would have occurred in the first iteration, causing the node to be excluded from the effective fixed-point nodes set.

Algorithm 5 Determining the set of fixed-point nodes

```
procedure getFixedPointNodes($G_D$, $G_S$, $s$, $V_{\text{focus}}$)
    $\zeta \leftarrow \emptyset$                    ▷ Mutated by applications of DfsWalk
    for all $v_{\text{focus}} \in V_{\text{focus}}$ do
        if $v_{\text{focus}}(s) \notin \zeta$ then
            DfsWalk($G_D, G_S, s, v_{\text{focus}}, \zeta$)
        end if
    end for
    return $\zeta$
end procedure
procedure DfsWalk($G_D, G_S, s, v_{\text{focus}}, \zeta$)
    $\zeta \leftarrow \zeta \cup \{v_{\text{focus}}(s)\}$
    if $|\mathcal{R}(G_S, s)| > 0$ then
        for all $v_{\text{value}} \in$ getValueNodes($G_D, G_S, s, v_{\text{focus}}$) do
            for all $s_{\text{ref}} \in \mathcal{R}(G_S, s)$ do
                if $v_{\text{value}}(s_{\text{ref}}) \notin \zeta$ then
                    DfsWalk($G_D, G_S, s_{\text{ref}}, v_{\text{value}}, \zeta$)
                end if
            end for
        end for
    end if
end procedure
```

Algorithm We assume a function *isRecursive* that receives a shapes graph and shape, and returns true if and only if the shape is recursive, i.e. if there is a path from and to that shape in the dependency graph. This can be determined by means of reachability query using a modified version of DFS that attempts to find the respective node. We assume a function *getFixedPointShapes* that receives a shapes graph and shape, and returns the reference closure of the given shape, i.e. all the shapes reachable from the recursive shape in the dependency graph. We assume a function *isReferencing* that receives a constraint component IRI and returns a boolean indicating whether the constraint component is a referencing constraint component.

The algorithm is given by Algorithm 6.

Step 1: Fixed-point Algorithm We extend the validation of a set of focus nodes against a shape (Algorithm 6) to use the fixed-point algorithm if (line 2) the shape is recursive. The set of effective fixed-point nodes is determined on line 10, and line 15 excludes non-referencing constraints after the first iteration.

A fixed-point node v_{fp} is non-conforming to a fixed-point shape s_{fp} if (line 20) the validation results of v_{fp} against the constraints of s_{fp} contain an erroneous validation result (by a non-referencing constraint) or a false conforming result (by a referencing constraint) for v_{fp}. If it is not non-conforming and the validation results contain an unknown conforming result then the conformance is unknown. Else (line 22) it must be conforming as all non-referencing constraints are satisfied and all referencing constraints (potentially none) declare conformance. This corresponds to the semantics in Table 4.2, in particular to the $\wedge$ operator, as shapes can be considered a conjunction of constraints.

Step 2: Generating Validation Results The validation results are generated by performing a final validation (line 34) of the non-conforming focus nodes (line 30) against the recursive shape. The validation is performed with the minimal fixed-point assignment σ of the previous step and without the boolean flag b, indicating that referencing constraints should avoid nested validation, and instead, use the assignment to produce validation results.

Algorithm 6 Validation of a set of focus nodes against a recursive shape

```
procedure validateNodesAgainstShape($G_D, G_S, V_{\text{focus}}, s, \sigma = \text{null}$)
  if $\sigma = \text{null} \land$ isRecursive($G_s, s$) then
    $\sigma \leftarrow \emptyset$
    $\zeta \leftarrow$ getFixedPointNodes($G_D, G_S, s, V_{\text{focus}}$)
    $S_{\text{fp}} \leftarrow$ getFixedPointShapes($G_S, s$)
    $b_{\text{fp}} \leftarrow$ True
    do
      $\sigma' = \sigma$
      for all $s_{\text{fp}} \in S_{\text{fp}}$ do
        $V_{\text{fp}} \leftarrow \{v_{\text{fp}} \mid v_{\text{fp}} \in \zeta(s_{\text{fp}}) \land s_{\text{fp}}(v_{\text{fp}}) \notin \sigma' \land \neg s_{\text{fp}}(v_{\text{fp}}) \notin \sigma'\}$
        if $|V_{\text{fp}}| > 0$ then
          $R \leftarrow \emptyset$
          for all $c \in$ getConstraints($G_S, s_{\text{fp}}$) do
            $(s', C, P) \leftarrow c$
            if $b_{\text{fp}} \lor$ isReferencing($c_2$) then
              $R \leftarrow R \cup$ validateNodesAgainstConstraint($G_D, G_S, V_{\text{fp}}, c, \sigma$, True)
            end if
          end for
          for all $v_{\text{fp}} \in V_{\text{fp}}$ do
            if $\left|\left\{ r \;\middle|\; \begin{array}{l} (r, \text{rdf:type}, \text{sh:ValidationResult}) \in R \\ \land\, (r, \text{sh:focusNode}, v_{\text{fp}}) \in R \end{array} \right\}\right| > 0 \lor (s_{\text{fp}}, \text{rsh:false}, v_{\text{fp}}) \in R$ then
              $\sigma = \sigma \cup \neg s_{\text{fp}}(v_{\text{fp}})$
            else if $(s_{\text{fp}}, \text{rsh:unknown}, v_{\text{fp}}) \notin R$ then
              $\sigma = \sigma \cup s_{\text{fp}}(v_{\text{fp}})$
            end if
          end for
        end if
      end for
      $b_{\text{fp}} \leftarrow$ False
    while $\sigma \neq \sigma'$
    $V_{\text{focus}} \leftarrow \{v_{\text{focus}} \mid v_{\text{focus}} \in V_{\text{focus}} \land \neg s(v_{\text{focus}}) \in \sigma\}$
  end if
  $R \leftarrow \emptyset$
  for all $c \in$ getConstraints($G_S, s$) do
    $R \leftarrow R \cup$ validateNodesAgainstConstraint($G_D, G_S, V_{\text{focus}}, c, \sigma$)
  end for
  return $R$
end procedure
```

Constraint Components The validators of non-referencing constraint components are identical to those in Section 6.2.2, and therefore, only produce validation results using solely the value nodes. The validators of referencing constraint components handle three modes:

1. When the validator is called without assignment σ and without boolean flag b (from outside the fixed-point algorithm), indicating that the constraint-declaring shape is non-recursive and that the validator must produce validation results, potentially using nested validation.

2. When the validator is called by the fixed-point algorithm with the assignment σ and boolean flag b true, indicating that the constraint-declaring shape is recursive and that the validator must produce conformance results by means of the dedicated vocabulary using σ, without nested validation.

3. When the fixed-point algorithm has reached a fixed point and the validator is called one

last time in order to generate a SHACL-like validation report. This time only for non-conforming focus nodes, with the minimal fixed-point assignment σ and without boolean flag b, indicating that the validator must produce validation results, potentially using σ.

The validation against referencing constraints components requiring conformance checking (e.g., *NodeConstraintComponent*) produces validation results based on whether value nodes conform to the referenced shapes. In the third mode, the validators of these constraint components do not perform nested validation, but instead, always use σ.

The validation results of a set of focus nodes against referencing constraint components requiring validation (i.e., *PropertyConstraintComponent*) is defined as the nested validation of the set of value nodes as focus nodes against the referenced shapes. So in the third mode, the validators of these constraint components still have to perform nested validation to produce their own validation results. For non-trivial patterns, this recursion could still be infinite, even though the focus node is non-conforming. A necessary condition is that the referenced shape is still recursive when only considering references by referencing constraint components requiring validation. A minimal instance is given by Example 33. Therefore, a recursion guard in mode three of the validators of referencing constraint components requiring validation is still needed to avoid infinite recursion.

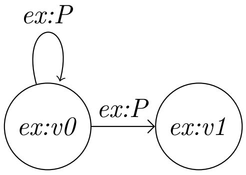
Appendix 1.3 describes and provides algorithms for the validation of one constraint component of each type for recursive SHACL.

Example 33. For the following shapes and data graph, producing the validation results of the non-conforming target node *ex:v0* against shape *ex:s0* results in infinite recursion without recursion guard in mode three of the validator of *PropertyConstraintComponent*:

```
ex:s0
  a sh:NodeShape ;
  sh:targetNode ex:v0 ;
  sh:property ex:s0P .

ex:s0P
  a sh:PropertyShape ;
  sh:path ex:P ;
  sh:property ex:s0P ;
  sh:minCount 1 .
```

Shapes graph



Data graph

△

Chapter 7

Experiments

7.1 Introduction

This chapter describes the performance and scalability experiments of our new native algorithm for validating non-recursive SHACL, proposed in Section 6.2.2, and our new native hybrid algorithm for validating recursive SHACL, proposed in Section 6.2.3.

We analyze the validation times of increasing data graphs against fixed recursive and non-recursive shapes graphs. For recursive shapes graphs, we analyze the time spent generating fixed-point nodes, assigning fixed-point shapes to nodes, and generating a SHACL-like validation report. Furthermore, we report the number of violating focus nodes and the number of iterations needed to reach a fixed point. Validation against non-recursive shapes graphs using the hybrid algorithm correspond to validation using the algorithm proposed in Section 6.2.2. Averages for each data and shapes graph are taken over the outcome of ten experiments.

Implementation The hybrid algorithm has been implemented in TopBraid's SHACL API[1] v1.3.0 and uses an in-memory repository by Apache Jena[2] v3.11.0. The implementation has explicitly been left single threaded. The source code has been made publicly available at `https://github.com/ChrisLahaye/shacl` to contribute to the open-source and academic community, as well as the data and shapes graphs used in our experiments.

The general approach of validation in TopBraid's SHACL API v1.3.0 corresponds to the algorithm proposed in Section 6.2.2, with the exception of how validation results are exchanged. In the implementation, constraint executors extend a shared graph with validation results, whereas our algorithms return the validation results as output.

Setting The experiments have been conducted on an Amazon Linux server (EC2 instance type r5.4xlarge[3]) hosted on Amazon Web Services. The server is equipped with 128GB of memory and 16 vCPU (a thread of an Intel Xeon Platinum 8175 processor). OpenJDK v11.0.5 is used as Java platform. Experiments are ran consecutively by spawning a new process when the previous one exits.

Data Graphs DBpedia is a project that extracts structured content from Wikipedia[4] and represents it in RDF. We use the English DBpedia 2016-10 data sets[5]: Instance Types, Labels, Mappingbased Literals, Mappingbased Objects, and Person data. Links to these data sets have been included with the source code of the implementation. The complete data set contains roughly

[1] `https://github.com/TopQuadrant/shacl`

[2] `https://jena.apache.org/`

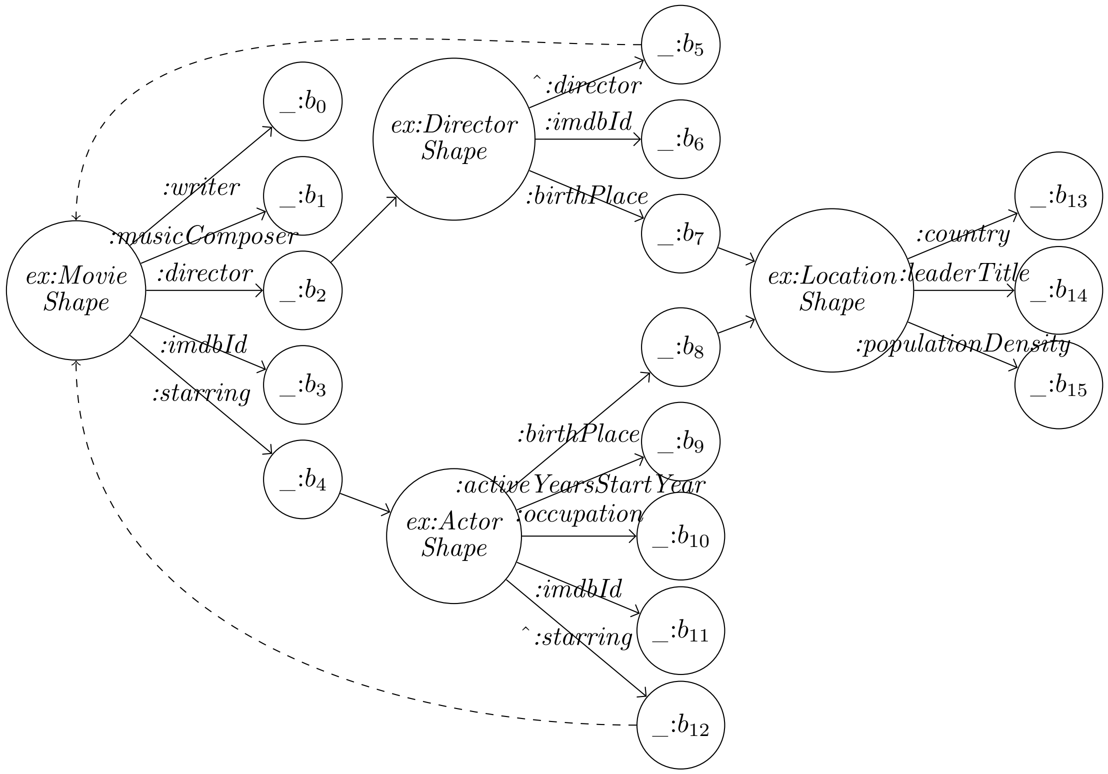
[3] `https://aws.amazon.com/ec2/instance-types/`

[4] `https://en.wikipedia.org/`

[5] `https://wiki.dbpedia.org/downloads-2016-10`

61.5 million triples. We validate against the complete data set, referred to by DBP_{100}, and against random samples of 10, 20, 40, and 80 percent of the triples, referred to by DBP_{10}, DBP_{20}, DBP_{40}, and DBP_{80}, respectively. Samples are taken once using GNU shuf[6] and are reused among experiments.

Shapes Graphs The shapes graphs have been based on the constraint sets of [13]. They are based on the patterns found in the DBpedia data set and test a variety of SHACL's features (e.g., non-referencing constraints, referencing constraints requiring conformance checking, referencing constraints requiring validation, and recursion). Firstly, we discuss the shapes in each shapes graph, secondly, the constraints declared by each shape and how they differ among shapes graphs, lastly, the applied changes to the original shapes. All shapes graphs have been included with the source code of the implementation. The dependency graphs of the greatest non-recursive and recursive shapes graphs ($nonRec_4$ and rec_4) have been visualized in Figure 7.1.



Dashed edges are only present in the recursive shapes graph. All edges are positive edges and have, instead of their parity, been labeled with the SPARQL property path of the (property) shape that the edge is pointing to. The namespace IRI `http://dbpedia.org/ontology/` is abbreviated by the prefix : instead of *dbo:* due to space limitations.

The SPARQL property paths ^*:director* and ^*:starring* are inverse paths. The values of an inverse path ^p for a node s and property path p are the nodes that have s as value for property path p. Simply stated, the values of ^*:starring* for an actor are all the movies that this actor is starring in.

Figure 7.1: The dependency graphs of rec_4 and $nonRec_4$

[6] `https://www.gnu.org/software/coreutils/manual/html_node/shuf-invocation.html`

The constraints have been divided into two sets, a non-recursive (*nonRec*) and recursive (*rec*) set. Each complete set contains the shapes MovieShape, ActorShape, LocationShape, and DirectorShape, and is further divided into the subsets nonRec_2, nonRec_3, and nonRec_4 (simlarly for *rec*) that contain the first two, first three, and all four shapes, respectively. This has been summarized in Table 7.1.

Table 7.1: Shapes in each shapes graph

Shapes graph	**Movie Shape**	**Actor Shape**	**Location Shape**	**Director Shape**
nonRec$_2$	X	X		
nonRec$_3$	X	X	X	
nonRec$_4$	X	X	X	X
rec$_2$	X	X		
rec$_3$	X	X	X	
rec$_4$	X	X	X	X

Having discussed the shapes in each shapes graph, we now discuss the constraints declared by each shape and how they differ among shapes graphs. The shape MovieShape targets instances of class *dbo:Film* and requires nodes to have at least one writer, music composer, director that conforms to shape DirectorShape (only in nonRec_4 and rec_4), and star that conforms to shape ActorShape, and exactly one Internet Movie Database (IMDb) ID. The shape ActorShape requires nodes to have actor as occupation, at least one active years start year, IMDb ID, and birth place that conforms to shape LocationShape (only in nonRec_{3+} and rec_{3+}), and lastly, all nodes having these actor nodes as star are required to conform to shape MovieShape (only in rec_{2+}). The shape LocationShape in nonRec_{3+} and rec_{3+} requires nodes to have at least one country, leader title, and population density. The shape DirectorShape in nonRec_4 and rec_4 requires nodes to have at least one birth date, active years start year, IMDb ID, and birth place that conforms to shape LocationShape, and all nodes having these director nodes as director are required to conform to the shape MovieShape (only in rec_4).

The shapes in [13] have been defined in a JSON serialization of the abstract syntax and in the concrete SHACL language. The recursive set defined in the abstract syntax is not strictly stratified, unless it is redefined in our new fragment strictly stratified $\mathcal{L}^+$. We made the following changes to the original shapes:

- Constraints of the form $\geq_n r.s$ for $n \geq 0$, property path r, and shape s were mapped to:

```
[
  sh:path r ;
  sh:qualifiedValueShape [ sh:node s ] ;
  sh:qualifiedMinCount n
] .
```

  For each focus node v, there must exist at least one value node v' that conforms to the referenced shape that is used as value for *sh:qualifiedValueShape*. The referenced shape declares a constraint of kind *NodeConstraintComponent* which requires that every value node for focus node v' conforms to the shape s. Since the constraint-declaring shape is a node shape, the set of value nodes is the set with as only member the focus node v'. So a value node v' conforms to the referenced shape, when it conforms to shape s. Therefore, a focus node v conforms to the shape if there exists at least one value node that conforms to shape s. We can simplify the validation by using shape s directly as value for *sh:qualifiedValueShape*.

  We replace the constraint parameter `sh:qualifiedValueShape [ sh:node s ]` by the semantically equivalent parameter `sh:qualifiedValueShape s` to eliminate redundant shapes.

- Constraints of the form $\geq_1 r.I$ for property path r and IRI I were mapped to:

```
[
  sh:path r ;
  sh:qualifiedValueShape [ sh:hasValue I ] ;
  sh:qualifiedMinCount 1
] .
```

For each focus node v, there must exist at least one value node v' that conforms to the referenced shape that is used as value for *sh:qualifiedValueShape*. The referenced shape declares a constraint of kind *HasValueConstraintComponent* which requires that one of the value nodes is I. Since the constraint-declaring shape is a node shape, the set of value nodes is the set with as only member the focus node v'. So a value node v' conforms to the referenced shape when v' is I. Therefore, a focus node v conforms to the shape if there exists at least one value node that is I. We can simplify the validation by using a constraint of kind *HasValueConstraintComponent*.

We replace the constraint of kind *QualifiedMinCountConstraintComponent* by a semantically equivalent constraint of kind *HasValueConstraintComponent* to eliminate unnecessary referencing constraints. We now have the shape:

```
[
  sh:path r ;
  sh:hasValue I
] .
```

- The original subsets $nonRec_4$ and rec_4 introduce the shape DirectorShape. This shape declares a property constraint with as value a property shape that declares a referencing constraint referencing the shape LocationShape. However, $nonRec_4$ declares a constraint of kind *NodeConstraintComponent*, i.e. $\neg(\geq_1 \textit{dbo:birthPlace}.\neg\text{LocationShape})$, and rec_4 of kind *QualifiedMinCountConstraintComponent*, i.e. $\geq_1 \textit{dbo:birthPlace}.\text{LocationShape}$. The prior requires all value nodes to conform to the shape while the latter only a single value node. We replace the prior by the latter to align the non-recursive and recursive shapes graphs.

- Property constraints with as value a property shape that declare a single constraint of kind *MinCountConstraintComponent*, i.e. $\geq_n r.\top$ for some $n \geq 0$ and property path r, that were only present in rec_4 but not in $nonRec_4$ have been removed to align the non-recursive and recursive shapes graphs.

When a new shape is referenced, for example by comparing $nonRec_2$ to $nonRec_3$ or $nonRec_3$ to rec_3, then a minimum cardinality constraint is replaced by a referencing constraint (*NodeConstraintComponent* or *QualifiedMinCountConstraintComponent*) referencing the new shape. So, each shape now declares the same pairs of property shapes and paths among subsets in which that shape is present.

Table 7.2 lists, for each data and shapes graph, the number of focus nodes identified for the validation against each node shape. These number have been determined using Algorithm 5 for the only target-declaring shape MovieShape and the nodes targeted by this shape. This procedure determines which nodes need to be validated against which shapes in order to conclude the conformance of the target nodes against the shape MovieShape.

Table 7.2: Number of focus nodes identified for the validation against each node shape

Node shape	**DBP$_{10}$**	**DBP$_{20}$**	**DBP$_{40}$**	**DBP$_{80}$**	**DBP$_{100}$**
nonRec$_{2}$					
MovieShape	11,233	22,370	44,656	89,677	111,938
ActorShape	2,945	9,552	26,088	61,496	79,438
nonRec$_{3}$					
MovieShape	11,233	22,370	44,656	89,677	111,938
ActorShape	2,945	9,552	26,088	61,496	79,438
LocationShape	427	1,688	5,140	11,915	14,641
nonRec$_{4}$					
MovieShape	11,233	22,370	44,656	89,677	111,938
ActorShape	2,945	9,552	26,088	61,496	79,438
LocationShape	506	1,974	6,012	13,591	16,627
DirectorShape	879	3,121	8,682	21,372	27,583
rec$_{2}$					
MovieShape	20,854	52,380	92,241	123,830	131,195
ActorShape	5,731	20,151	46,632	84,260	99,020
rec$_{3}$					
MovieShape	20,854	52,380	92,241	123,830	131,195
ActorShape	5,731	20,151	46,632	84,260	99,020
LocationShape	720	2,846	7,227	13,886	16,253
rec$_{4}$					
MovieShape	23,528	59,758	102,882	134,761	141,690
ActorShape	6,233	21,626	48,621	85,391	99,357
LocationShape	871	3,379	8,378	15,605	18,188
DirectorShape	1,587	5,769	13,376	25,289	30,090

7.2 Results

Figure 7.2 shows a visualization of the average validation time versus the data graph. Table 7.3 lists the average validation time of validating each data graph against every shapes graph, as well as the number of violating focus nodes and a breakdown for recursive shapes into the average time spent generating fixed-point nodes, assigning fixed-point shapes to nodes, generating a SHACL-like validation report, and the number of iterations needed to reach a fixed point. Figure 7.3 shows a visualization of the breakdown for recursive shapes graphs.

The general observation is that the total validation time increases with the size of the data graph (see Figure 7.2) since the number of target, focus, and value nodes increases as the data graph becomes more complete (see Table 7.2). The validation time against the recursive data graphs increase faster and take longer in total compared to the non-recursive data graphs due to the cost of handling recursion.

A second observation is that this trend is less or not present at all when comparing DBP_{100} to DBP_{80} for any recursive shapes graph. An increase in the size of the data graph comes with an increase in the number of fixed-point nodes (see Table 7.2) and time spent generating these (see Figure 7.3). An increase in fixed-point nodes typically results in an increase in time spent assigning fixed-point shapes to these nodes. However, this does not necessarily have to be the case as changes in the number of fixed-point and value nodes could also significantly reduce the time needed to decide that a constraint has been satisfied or violated. This is the case for DBP_{100}

compared to DBP_{80} and any recursive shapes graph, caused by new value nodes causing a violation against the constraint declared by *ex:ActorShape* of kind *NodeConstraintComponent* referencing *ex:MovieShapes*, i.e. $\forall$ˆ*dbo:starring*.MovieShape, to be determined sooner.

The time spent generating a SHACL-like validation report increases with the size of the data graph (see Figure 7.3) as more target nodes are being identified (see Table 7.2), while a majority of them still violate the shapes that target them. This happens when the data graph is still not complete enough or when the shapes graph does not correspond well enough to the patterns in the data set. In our case its the latter, as validating the complete data set indicates that 11,233 of the 111,938 targeted nodes are still violated.

The final observation is that validating DBP_{100} against $nonRec_4$ and rec_4 take on average 54.8 and 56.4 seconds, respectively. One of the key aspects is that the cost of handling recursion in the complete data set is only marginal. This demonstrates the effectiveness of our pruning strategies and minimal overhead by running punctual fixed-point iterations.

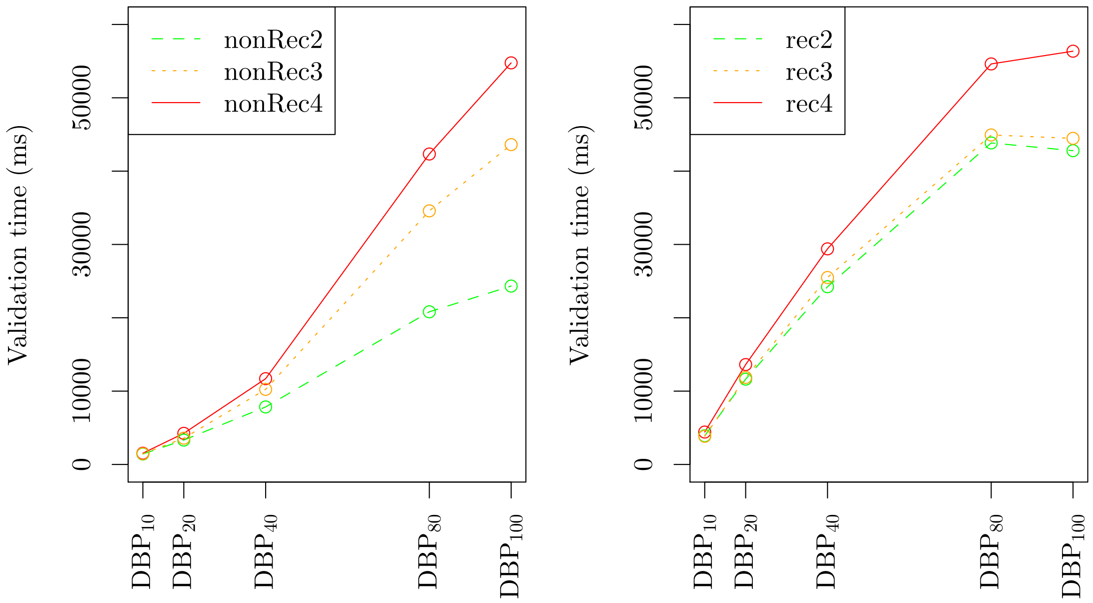
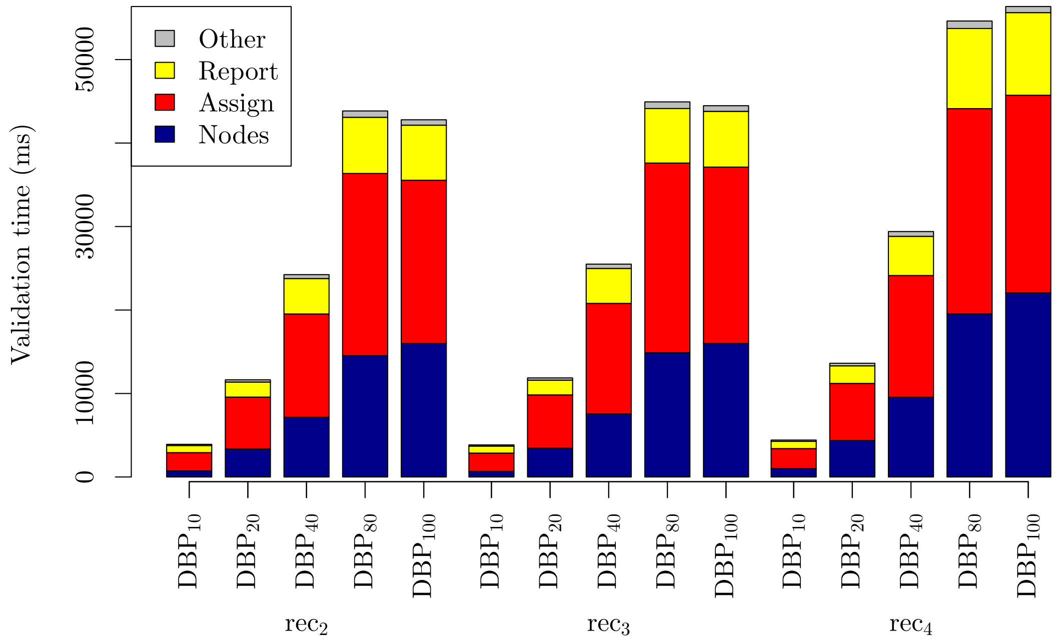


Figure 7.2: Average validation times vs. data graph

Table 7.3: Average validation times and times spent for Nodes, Assign, Report, and Other

The table below shows, in order, the data graph, shapes graph, average total validation time (Total), and average time spent generating fixed-point nodes (Nodes), assigning fixed-point shapes to nodes (Assign), generating a SHACL-like validation report (Report), and doing other work (Other), i.e. the remaining difference with the total validation time, as well as the number of violating focus nodes (Viola.) and the number of iterations needed to reach a fixed point (Iter.). All times are reported in seconds, with the exception of the value not applicable (NA) for Nodes, Assign, Report, and Other when the shapes graph is non-recursive.

Data Graph	**Shapes Graph**	**Total**	**Nodes**	**Assign**	**Report**	**Other**	**Viola.**	**Iter.**
DBP$_{10}$	nonRec$_2$	1.4	NA	NA	NA	NA	11,233	NA
DBP$_{20}$	nonRec$_2$	3.3	NA	NA	NA	NA	22,370	NA
DBP$_{40}$	nonRec$_2$	7.8	NA	NA	NA	NA	44,655	NA
DBP$_{80}$	nonRec$_2$	20.8	NA	NA	NA	NA	89,556	NA
DBP$_{100}$	nonRec$_2$	24.3	NA	NA	NA	NA	111,113	NA
DBP$_{10}$	nonRec$_3$	1.5	NA	NA	NA	NA	11,233	NA
DBP$_{20}$	nonRec$_3$	3.6	NA	NA	NA	NA	22,370	NA
DBP$_{40}$	nonRec$_3$	10.2	NA	NA	NA	NA	44,656	NA
DBP$_{80}$	nonRec$_3$	34.6	NA	NA	NA	NA	89,653	NA
DBP$_{100}$	nonRec$_3$	43.6	NA	NA	NA	NA	111,629	NA
DBP$_{10}$	nonRec$_4$	1.5	NA	NA	NA	NA	11,233	NA
DBP$_{20}$	nonRec$_4$	4.2	NA	NA	NA	NA	22,370	NA
DBP$_{40}$	nonRec$_4$	11.7	NA	NA	NA	NA	44,656	NA
DBP$_{80}$	nonRec$_4$	42.3	NA	NA	NA	NA	89,675	NA
DBP$_{100}$	nonRec$_4$	54.8	NA	NA	NA	NA	111,883	NA
DBP$_{10}$	rec$_2$	3.9	0.7	2.2	0.9	0.1	11,233	3
DBP$_{20}$	rec$_2$	11.6	3.3	6.2	1.8	0.3	22,370	3
DBP$_{40}$	rec$_2$	24.2	7.2	12.4	4.2	0.5	44,656	3
DBP$_{80}$	rec$_2$	43.8	14.5	21.8	6.7	0.8	89,674	4
DBP$_{100}$	rec$_2$	42.8	16.0	19.5	6.6	0.6	111,919	4
DBP$_{10}$	rec$_3$	3.8	0.7	2.2	0.8	0.1	11,233	3
DBP$_{20}$	rec$_3$	11.9	3.5	6.4	1.8	0.3	22,370	3
DBP$_{40}$	rec$_3$	25.5	7.5	13.3	4.2	0.5	44,656	3
DBP$_{80}$	rec$_3$	44.9	14.9	22.7	6.5	0.8	89,677	4
DBP$_{100}$	rec$_3$	44.5	16.0	21.1	6.7	0.7	111,927	4
DBP$_{10}$	rec$_4$	4.4	1.0	2.4	0.9	0.2	11,233	3
DBP$_{20}$	rec$_4$	13.6	4.4	6.8	2.1	0.3	22,370	3
DBP$_{40}$	rec$_4$	29.4	9.6	14.6	4.7	0.6	44,656	3
DBP$_{80}$	rec$_4$	54.6	19.5	24.6	9.6	0.9	89,677	3
DBP$_{100}$	rec$_4$	56.4	22.0	23.7	9.9	0.7	111,938	4



This figure shows the data graph, recursive shapes graph, and the average time spent generating fixed-point nodes (Nodes), assigning fixed-point shapes to nodes (Assign), generating a SHACL-like validation report (Report), and doing other work (Other).

Figure 7.3: Average times spent for Nodes, Assign, Report, and Other

7.3 Conclusions

As expected, total validation time increases with the number of shapes in the shapes graph and the size of the data graph as a greater graph identifies more target, focus, and value nodes. This correlation is stronger for recursive shapes graphs. The number of target nodes is the most significant predictor of the validation time for both non-recursive and recursive shapes graphs. The validation time for non-recursive shapes graphs is more significantly impacted by changes in the number of shapes compared to recursive shapes graphs. The largest portion of the validation time for recursive shapes graphs is spent assigning fixed-point shapes to nodes.

For the complete data graph and non-recursive shapes graphs, adding the shape LocationShape by comparing $nonRec_3$ to $nonRec_2$ results in a 79% increase in validation time. Adding the shape DirectorShape by comparing $nonRec_4$ to $nonRec_3$ results in another 25% increase. Validation against *nonRec*4 takes on average 54.8 seconds. For the complete data graph and recursive shapes graphs, adding the shape LocationShape by comparing rec_3 to rec_2 results in a 4% increase in validation time. Adding the shape DirectorShape by comparing rec_4 to rec_3 results in another 27% increase. Validation against *rec*4 takes on average 56.4 seconds.

Our experiments indicate that it is indeed possible to validate large real-world data sets against complex shapes graphs in short periods of time, in the order of seconds. Even selectively adding recursion did not have a significant impact on validation time. In fact, our experiments only showed a 3% increase in total validation time, demonstrating the effectiveness of our pruning strategies and minimal overhead by running punctual fixed-point iterations.

7.4 Discussion

Corman et al. validated the same data set against a similar non-recursive SHACL schema by means of a single SPARQL query in 5.3 seconds [13]. This demonstrates that large real-world data sets can be validated against non-recursive shapes graphs using a single SPARQL query in an even shorter period of time, without extra in-memory computation. This approach requires access to a SPARQL endpoint. Using a SPARQL endpoint allows validation to be performed outside the database engine from an external system. Whereas our approach assumes an in-memory repository, although this is not required for the taken approach to work. Important to note is that there are many differences between both approaches making a comparison non-trivial. Our algorithms have explicitly been left single-threaded, whereas as a SPARQL engine most likely performs multi-threaded evaluation.

SPARQL does not support recursion unless the recursive SPARQL extension, introduced in [28], is used. Therefore, only non-recursive schemas can be expressed in SPARQL. Validation of recursive schemas is based on a minimal fixed-point assignment. Finding such an assignment with respect to a set of SHACL constraints is the problem solved in our study and in [13]. Our study is aimed at tractability and takes a more practical view towards validating SHACL, and therefore, our algorithms use the concrete SHACL language and are only sound and complete for tractable SHACL fragments; whereas the approach in [13] works for arbitrary SHACL schemas defined in the abstract syntax, although at the cost of intractable.

The approach in [13] consists of a combination of a SPARQL-based and rule-based approach to validate recursive SHACL. SPARQL is used to retrieve the target nodes for each shape and all value nodes needed for a node to satisfy the shape's constraints. This is similar to the generation of the fixed-point nodes in our algorithm. The query result is used to generate a set of rules of the form of the form $l_0 \wedge ... \wedge l_n \implies s(v)$, where l_i is either $s_i(v_i)$ or $\neg s_i(v_i)$, for some shape s_i and node v and v_i. Additional rules are add to ensure shapes (the consequent) can only be assigned if one of the constraints (antecedent) holds and that target nodes can only be assigned the positive literal. Then solutions to these rules are found using a SAT solver. The problem of finding a satisfying assignment to these rules has worst-case exponential complexity, hence the intractability. They also showed that when the fragment is tractable, that the SAT solver can be replaced with on-the-fly inferencing which runs in polynomial time. This is similar to the approach of our algorithm of finding a minimal fixed-point assignment σ_{minFix} and determining that no negative shape label is being assigned.

In our attempt to validate our results with another validator, we observed that the results obtained by the implementation [12] of the algorithm of [13] were incorrect. The results obtained by both validators did match when we replaced a universal quantification by an existential quantification. This might be related to the fact that the schema is actually not tractable under the fragments considered in [13], while their implementation with in-memory inference assumes a tractable fragment. The schema is only tractable when considering the strictly stratified $\mathcal{L}^+$ fragment introduced in this thesis, which may have been done implicitly.

Our algorithm has been implemented in TopQuadrant's SHACL API, with as result that our obtained results are impacted by the used functionality of TopQuadrant's SHACL API (e.g., determining the values of a SHACL property path). In order to form a more accurate opinion about the performance of our algorithm, the algorithm needs to be implemented stand-alone while revisiting the implementation of critical functionality. Finally, adding multi-threading to our algorithms would allow for significant performance optimizations, for instance, by validating constraints concurrently.

Chapter 8

Conclusions

8.1 Summary

We researched various approaches to validate RDF graphs. Using SPARQL for validation is an interesting choice due to its wide adoption. Using a grammar-based approach for validation like SHACL allows constraints to be expressed in a concise and declarative manner, this in contrast to SPARQL. We mostly focused on the constraint language SHACL as it can be seen as the most prominent constraint language due to its standardization, support for recursion, and ability to validate and declare high-level reusable components of arbitrary SPARQL-based constraints.

Validation of SHACL Core ($\mathcal{L}$) is NP-complete. Intractability stems from the ability to validate constraints using arbitrary negation and recursion. This property still holds for the severely limited fragment stratified $\mathcal{L}_{\geq 1,\neg,\wedge}$ with stratified negation and just basic operators. Two tractable recursive fragments have been identified in previous research: $\mathcal{L}_{\geq n,\wedge,\vee,r,\mathrm{EQ}}$ which has an additional native operator for disjunction but disallows negation, and strictly stratified $\mathcal{L}$ with strictly stratified negation.

We proposed a new tractable and more expressive recursive SHACL fragment, called strictly stratified $\mathcal{L}^{+}$, with strictly stratified negation and additional native operators for universal quantification and disjunction. It includes $\mathcal{L}_{\geq n,\wedge,\vee,r,\mathrm{EQ}}$ and strictly stratified $\mathcal{L}$, and allows for additional constraints to be expressed using native operators to express universal quantification and disjunction without the use of negation. This implies that use of such constraints is not constrained by the strict stratification restrictions. Validation of strictly stratified $\mathcal{L}^{+}$ is P-complete in combined complexity.

Validation of recursive schemas is based on a minimal fixed-point assignment. We proved that validation of all tractable recursive fragments identified so far, in particular strictly stratified $\mathcal{L}^{+}$, is indeed tractable. We showed that if the minimal fixed-point assignment, which can be computed in polynomial time in $|G|+|S|$, does not assign the negated shape to a node targeted by this shape, that there must exist another constraint satisfying assignment that successfully assigns the shape to this node. Meaning that a node is valid against a recursive shape if and only if the minimal fixed-point assignment does not assign the negated shape.

We studied the differences between validating non-referencing constraints using a native implementation and by means of a SPARQL query. Evaluation of SPARQL queries is impacted by the query plan generator and optimizer. The queries analyzed in this study resulting from mapping SHACL constraints to SPARQL queries are simple queries due to the validation strategy where each constraint is validated by a single query. Validation of a single constraint is a simple procedure for which parsing and optimizing queries may not give any additional benefit justifying the induced overhead. We assessed validation performance by reasoning about the number of index lookups and scanned triples for SPARQL query plans generated using Amazon Neptune and potential native implementations. We find that less than property pair constraints may benefit from a native implementation as our native implementation has a constant number of index lookups compared to a linear number for SPARQL-based validation in Amazon Neptune. Minimum and maximum cardinality constraints may as well benefit from a native implementation as the number of scanned triples can be bound based on the cardinality value, whereas SPARQL-based validation in Amazon Neptune scans all triples. We identified that SPARQL-based validation of these constraints may achieve that same benefit by limiting the number of extracted join partners.

We proposed a new native algorithm to validate non-recursive SHACL using immediate constraint evaluation. Our algorithm runs in polynomial time in $|G| + |S|$, operates on the concrete SHACL language, and generates SHACL-compliant validation reports. The validation against referencing constraints is based on the nested validation of nodes against referenced shapes. This recursion may be infinite when the referenced shape is recursive and the node part of a cycle.

We propose a new native hybrid algorithm to validate recursive SHACL and mitigate this problem by extending the algorithm for non-recursive SHACL with a minimal fixed-point algorithm handling validation against recursive shapes. Our algorithm is sound and complete for all tractable fragments identified so far, including non-recursive SHACL. It runs in polynomial time in $|G|+|S|$, operates on the concrete SHACL language, and generates SHACL-like validation reports.

Besides using a tractable fragment, our hybrid algorithm employs additional optimizations techniques to make validation more efficient. The dependency graph of the shapes graph is used to determine: whether a shape is recursive and which algorithm is needed; the minimal set of fixed-point shapes that need to be assigned to conclude the conformance against the recursive shape; and in what order fixed-point shapes should be assigned to minimize inconclusive answers. For each fixed-point shape, the data graph is iterated and a dedicated set of value nodes is constructed to which the fixed-point shape needs to be assigned to conclude the conformance against the recursive shape. Then fixed-point shapes are only assigned to effective fixed-point nodes, which are fixed-point nodes that do not have the shape or its negation assigned, utilizing the monotonicity property. Lastly, non-referencing constraints, i.e. constraints independent on the assignment, are deactivated in subsequent fixed-point iterations, hereby eliminating redundant validations.

Our experiments of the performance and scalability of both algorithms demonstrate that validation of large real-world data sets against complex SHACL shapes graphs can be performed efficiently, in the order of seconds. Even selectively adding recursion did not have a significant impact on validation time; our experiments only showed an increase of 3%. Hereby, demonstrating the effectiveness of our optimization techniques and minimal overhead by running punctual fixed-point iterations.

With our study, we advanced the state-of-the-art in graph validation by proposing a new tractable and more expressive recursive SHACL fragment and an effective method to validate tractable fragments. Hereby we assist in bringing structure to RDF graphs, a need that only increases as graph data is becoming more popular and constantly growing in terms of volume and variety. By doing so, we hope to have a positive impact on the evolution and adoption of graph database management systems, as well as on the meaningful data analyses that can be performed using graph data.

8.2 Future Work

Future work may identify new tractable recursive SHACL fragments, potentially by relaxing the strict stratification restrictions using for instance additional properties of the dependency graph; as well as new methods to efficiently validate SHACL.

Our algorithms could further be improved by utilizing multi-threading, for example to validate constraints concurrently. Another interesting future work proposal is to make them incremental. Hereby avoiding that a complete data set needs to be revalidated when the data set changes, but instead, only requiring validation of the part affected by the update. For a recursive shape, it may be possible to use the minimal fixed-point assignment of the latest validation, remove assignments to nodes affected by the update, and then run the fixed-point algorithm starting with this assignment.

Bibliography

[1] Serge Abiteboul. “Querying Semi-Structured Data”. In: *Proceedings of the 6th International Conference on Database Theory*. ICDT ’97. London, UK, UK: Springer-Verlag, 1997, pp. 1–18. ISBN: 978-3-540-62222-2. URL: `http://dl.acm.org/citation.cfm?id=645502.656103` (visited on 25/10/2019).

[2] Serge Abiteboul, Peter Buneman and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. ISBN: 978-1-55860-622-7.

[3] R. Ahlswede et al. “Network information flow”. In: *IEEE Transactions on Information Theory* 46.4 (July 2000), pp. 1204–1216. ISSN: 1557-9654. DOI: `10.1109/18.850663`.

[4] U. Alon. “Biological Networks: The Tinkerer as an Engineer”. In: *Science* 301.5641 (Sept. 2003), pp. 1866–1867. ISSN: 0036-8075, 1095-9203. DOI: `10.1126/science.1089072`. URL: `https://science.sciencemag.org/content/301/5641/1866` (visited on 25/11/2019).

[5] Renzo Angles and Claudio Gutierrez. “Survey of Graph Database Models”. In: *ACM Comput. Surv.* 40.1 (Feb. 2008), 1:1–1:39. ISSN: 0360-0300. DOI: `10.1145/1322432.1322433`. URL: `http://doi.acm.org/10.1145/1322432.1322433` (visited on 25/06/2019).

[6] Dörthe Arndt et al. “Using Rule-Based Reasoning for RDF Validation”. In: *Proceedings of the International Joint Conference on Rules and Reasoning*. Ed. by Stefania Costantini et al. Vol. 10364. Lecture Notes in Computer Science. Springer, July 2017, pp. 22–36. ISBN: 978-3-319-61252-2. DOI: `10.1007/978-3-319-61252-2_3`.

[7] Albert-László Barabási and Zoltán N. Oltvai. “Network biology: understanding the cell’s functional organization”. In: *Nature Reviews Genetics* 5.2 (Feb. 2004), pp. 101–113. ISSN: 1471-0064. DOI: `10.1038/nrg1272`. URL: `https://www.nature.com/articles/nrg1272` (visited on 25/11/2019).

[8] Peter S. Bearman and James Moody. “Suicide and Friendships Among American Adolescents”. In: *American Journal of Public Health* 94.1 (Jan. 2004), pp. 89–95. ISSN: 0090-0036. URL: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1449832/` (visited on 25/11/2019).

[9] Thomas Bosch et al. “The role of reasoning for RDF validation”. In: *Proceedings of the 11th International Conference on Semantic Systems - SEMANTICS ’15*. Vienna, Austria: ACM Press, 2015, pp. 33–40. ISBN: 978-1-4503-3462-4. DOI: `10.1145/2814864.2814867`. URL: `http://dl.acm.org/citation.cfm?doid=2814864.2814867` (visited on 19/06/2019).

[10] Julien Corman, Juan L Reutter and Ognjen Savkovic. “A Tractable Notion of Stratication for SHACL”. In: *The Semantic Web – ISWC 2018*. 2018, pp. 1–4.

[11] Julien Corman, Juan L. Reutter and Ognjen Savković. “Semantics and Validation of Recursive SHACL”. In: *The Semantic Web – ISWC 2018*. Ed. by Denny Vrandečić et al. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 318–336. ISBN: 978-3-030-00671-6.

[12] Julien Corman et al. “shacl2sparql: Validating a sparql Endpoint against Recursive shacl Constraints”. In: *The Semantic Web – ISWC 2019*. 2019, pp. 1–4.

[13] Julien Corman et al. “Validating Shacl Constraints over a Sparql Endpoint”. In: *The Semantic Web – ISWC 2019*. Oct. 2019, pp. 145–163. ISBN: 978-3-030-30792-9. DOI: 10.1007/978-3-030-30793-6_9.

[14] Dan Brickley and R.V. Guha. *RDF Schema 1.1*. Feb. 2014. URL: https://www.w3.org/TR/rdf-schema/ (visited on 24/10/2019).

[15] David Beckett et al. *RDF 1.1 Turtle*. Feb. 2014. URL: https://www.w3.org/TR/turtle/ (visited on 15/11/2019).

[16] B. A. Eckman and P. G. Brown. “Graph data management for molecular and cell biology”. In: *IBM Journal of Research and Development* 50.6 (Nov. 2006), pp. 545–560. ISSN: 0018-8646. DOI: 10.1147/rd.506.0545.

[17] Peter M. Fischer et al. “RDF Constraint Checking”. In: *EDBT/ICDT Workshops*. 2015.

[18] Jose Emilio Labra Gayo et al. “Validating RDF Data”. In: *Synthesis Lectures on the Semantic Web: Theory and Technology* 7.1 (Sept. 2017), pp. 1–328. ISSN: 2160-4711, 2160-472X. DOI: 10.2200/S00786ED1V01Y201707WBE016. URL: http://www.morganclaypool.com/doi/10.2200/S00786ED1V01Y201707WBE016 (visited on 11/06/2019).

[19] Stephan Grimm and Boris Motik. “Closed World Reasoning in the Semantic Web through Epistemic Operators.” In: *Proceedings of the OWLED*05 Workshop on OWL*. Jan. 2005.

[20] W3C SPARQL Working Group. *SPARQL 1.1 Overview*. Mar. 2013. URL: https://www.w3.org/TR/sparql11-overview/ (visited on 24/10/2019).

[21] Holger Knublauch. *DASH Data Shapes Vocabulary*. May 2018. URL: http://datashapes.org/dash.html (visited on 25/10/2019).

[22] Holger Knublauch. *SPIN Modeling Vocabulary*. Feb. 2011. URL: https://spinrdf.org/spin.html (visited on 25/10/2019).

[23] Holger Knublauch and Dimitris Kontokostas. *Shapes Constraint Language (SHACL)*. July 2017. URL: https://www.w3.org/TR/shacl/ (visited on 24/10/2019).

[24] Holger Knublauch. *From SPIN to SHACL*. Aug. 2017. URL: https://spinrdf.org/spin-shacl.html (visited on 19/01/2020).

[25] Georg Lausen, Michael Meier and Michael Schmidt. “SPARQLing Constraints for RDF”. In: *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT ’08. event-place: Nantes, France. New York, NY, USA: ACM, 2008, pp. 499–509. ISBN: 978-1-59593-926-5. DOI: 10.1145/1353343.1353404. URL: http://doi.acm.org/10.1145/1353343.1353404 (visited on 25/06/2019).

[26] Seth A. Myers et al. “Information Network or Social Network?: The Structure of the Twitter Follow Graph”. In: *Proceedings of the 23rd International Conference on World Wide Web*. WWW ’14 Companion. event-place: Seoul, Korea. New York, NY, USA: ACM, 2014, pp. 493–498. ISBN: 978-1-4503-2745-9. DOI: 10.1145/2567948.2576939. URL: http://doi.acm.org/10.1145/2567948.2576939 (visited on 25/11/2019).

[27] Jorge Pérez, Marcelo Arenas and Claudio Gutierrez. “Semantics and Complexity of SPARQL”. In: *ACM Trans. Database Syst.* 34.3 (Sept. 2009), 16:1–16:45. ISSN: 0362-5915. DOI: 10.1145/1567274.1567278. URL: http://doi.acm.org/10.1145/1567274.1567278 (visited on 25/06/2019).

[28] Juan L. Reutter, Adrián Soto and Domagoj Vrgoč. “Recursion in SPARQL”. In: *The Semantic Web - ISWC 2015*. Ed. by Marcelo Arenas et al. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 19–35. ISBN: 978-3-319-25007-6.

[29] Richard Cyganiak, David Wood and Markus Lanthaler. *RDF 1.1 Concepts and Abstract Syntax*. Feb. 2014. URL: https://www.w3.org/TR/rdf11-concepts/ (visited on 15/11/2019).

[30] Kunal Sengupta, Adila Alfa Krisnadhi and Pascal Hitzler. “Local Closed World Semantics: Grounded Circumscription for OWL”. In: *The Semantic Web – ISWC 2011*. Ed. by Lora Aroyo et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 617–632. ISBN: 978-3-642-25073-6. DOI: 10.1007/978-3-642-25073-6_39.

[31] Pedro Szekely et al. “Building and Using a Knowledge Graph to Combat Human Trafficking”. In: *The Semantic Web - ISWC 2015*. Ed. by Marcelo Arenas et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 205–221. ISBN: 978-3-319-25010-6. DOI: 10.1007/978-3-319-25010-6_12.

[32] Alfred Tarski et al. “A lattice-theoretical fixpoint theorem and its applications.” In: *Pacific journal of Mathematics* 5.2 (1955), pp. 285–309.

[33] Thomas Baker and Eric Prudhommeaux. *Shape Expressions (ShEx) Primer*. July 2017. URL: https://www.w3.org/TR/shex-primer/ (visited on 24/10/2019).

[34] Johan Ugander et al. “The Anatomy of the Facebook Social Graph”. In: *arXiv:1111.4503 [physics]* (Nov. 2011). arXiv: 1111.4503. URL: http://arxiv.org/abs/1111.4503 (visited on 25/11/2019).

[35] W3C OWL Working Group. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. Dec. 2012. URL: https://www.w3.org/TR/owl-overview/ (visited on 24/10/2019).

[36] Cathrin Weiss, Panagiotis Karras and Abraham Bernstein. “Hexastore: sextuple indexing for semantic web data management”. In: *Proceedings of the VLDB Endowment* 1.1 (Aug. 2008), pp. 1008–1019. ISSN: 2150-8097. DOI: 10.14778/1453856.1453965. URL: https://doi.org/10.14778/1453856.1453965 (visited on 14/01/2020).

Appendices

Appendix 1

Additional Algorithms for SHACL Constraint Components

1.1 Introduction

This appendix describes and provides validator implementations for one constraint component of each type considered in this thesis, i.e. referencing constraint component requiring validation, referencing constraint component requiring conformance checking, and non-referencing constraint components.

Abstractions We abstract away from the creation of validation results. Instead, we simply assume the existence of a function *createResult* that receives a constraint, focus node, value, and textual description, and generates the corresponding validation result as defined in [23]. This is a trivial task as all mandatory properties can easily be determined from the data available in a constraint validator.

We further assume a recursion guard (see Section 6.2.3) that keeps track of the nodes and shape being recursed. The function *startRec* receives a set of nodes and shape, returns the nodes that are not being recursed for the shape, and then registers them as being recursed. The function *endRec* receives a set of (recursed) nodes and shape, and unregisters the nodes for the shape as being recursed.

1.2 Immediate Constraint Evaluation

PropertyConstraintComponent Restricts the set of value nodes to have a given property shape (value of mandatory parameter *sh:property*). The validation results are the results of validating the set of value nodes as focus nodes against the property shape. This is a referencing constraint component requiring validation.

Algorithm 7 Validation of a set of focus nodes against a constraint of kind PropertyConstraintComponent

```
1: procedure validateNodesAgainstPropertyConstraint($G_D, G_S, V_{\text{focus}}, c$)
2:     $(s, C, P) \leftarrow c$
3:     $s_{\text{property}} \leftarrow P(\text{sh:property})$
4:     if isNodeShape($s$) then
5:         return validateNodesAgainstShape($G_D, G_S, V_{\text{focus}}, s_{\text{property}}$)
6:     else
7:         $R \leftarrow \emptyset$
8:         for all $v_{\text{focus}} \in V_{\text{focus}}$ do
9:             $V_{\text{value}} \leftarrow$ getValueNodes($G_D, G_S, s, v_{\text{focus}}$)
10:            $R \leftarrow R \cup$ validateNodesAgainstShape($G_D, G_S, V_{\text{value}}, s_{\text{property}}$)
11:        end for
12:        return $R$
13:    end if
14: end procedure
```

Note that the if-statement on line 4 is not required and that the negative branch is sufficient to handle both cases, when the constraint is declared by a node and property shape. The positive branch however limits the number of calls to *validateNodesAgainstShape* when the constraint has been declared by a node shape. The negative branch would call *validateNodesAgainstShape* for each focus node with a single value node as the value nodes for node shapes are the individual focus nodes forming a set with exactly one member [23], whereas the positive branch would make a single call with all the focus nodes as value nodes. This corresponds to the redefined definitions of validation to consider a set of focus nodes instead of a single focus node.

EqualsConstraintComponent Restricts the set of value nodes to equal the set of nodes that have the focus node as subject and the value of a given property (value of mandatory parameter *sh:equals*) as predicate . If a value node does not exist as a value of the given property, there is a validation result with the value node as value. If a value of the given property is not a value node, there exists a validation result with the value as value. This is a non-referencing constraint component.

Algorithm 8 Validation of a set of focus nodes against a constraint of kind EqualsConstraintComponent

```
procedure validateNodesAgainstEqualsConstraint($G_D, G_S, V_{\text{focus}}, c$)
    $(s, C, P) \leftarrow c$
    $p_{\text{equals}} \leftarrow P(\text{sh:equals})$
    $R \leftarrow \emptyset$
    for all $v_{\text{focus}} \in V_{\text{focus}}$ do
        $V_{\text{value}} \in$ getValueNodes($G_D, G_S, s, v_{\text{focus}}$)
        for all $v_{\text{value}} \in V_{\text{value}} \wedge (v_{\text{focus}}, p_{\text{equals}}, v_{\text{value}}) \notin G_D$ do
            $R \leftarrow R \cup$ createResult($c, v_{\text{focus}}, v_{\text{value}},$ ”not a value of equals”)
        end for
        for all $(v_{\text{focus}}, p_{\text{equals}}, v_{\text{value}}) \in G_S \wedge v_{\text{value}} \notin V_{\text{value}}$ do
            $R \leftarrow R \cup$ createResult($c, v_{\text{focus}}, v_{\text{value}},$ ”not a value node”)
        end for
    end for
    return $R$
end procedure
```

NodeConstraintComponent Restricts the set of value nodes to conform to the given node shape (value of mandatory parameter *sh:node*). If a value node does not conform to the node shape, there is a validation result with the value node as value. This is a referencing constraint component requiring conformance checking.

Algorithm 9 Validation of a set of focus nodes against a constraint of kind NodeConstraintComponent

```
procedure validateNodesAgainstNodeConstraint($G_D, G_S, V_{\text{focus}}, c$)
    $(s, C, P) \leftarrow c$
    $s_{\text{node}} \leftarrow P(\text{sh:node})$
    $R \leftarrow \emptyset$
    for all $v_{\text{focus}} \in V_{\text{focus}}$ do
        for all $v_{\text{value}} \in$ getValueNodes($G_D, G_S, s, v_{\text{focus}}$) do
            if $|$validateNodesAgainstShape($G_D, G_S, \{v_{\text{value}}\}, s_{\text{node}}$)$| > 0$ then
                $R \leftarrow R \cup$ createResult($c, v_{\text{focus}}, v_{\text{value}},$ ”does not conform to ” $+ s_{\text{node}}$)
            end if
        end for
    end for
    return $R$
end procedure
```

1.3 Recursive SHACL

PropertyConstraintComponent Restricts the set of value nodes to have a given property shape (value of mandatory parameter *sh:property*). The validation results are the results of validating the set of value nodes as focus nodes against the property shape. This is a referencing constraint component requiring validation.

Algorithm 10 Validation of a set of focus nodes against a constraint of kind PropertyConstraintComponent

```
procedure validateNodesAgainstPropertyConstraint($G_D, G_S, V_{\text{focus}}, c, \sigma = \text{null}, b = \text{False}$)
    $(s, C, P) \leftarrow c$
    $s_{\text{property}} \leftarrow P(\text{sh:property})$
    $R \leftarrow \emptyset$
    if isNodeShape($s$) then
        if $b$ then
            for all $v_{\text{focus}} \in V_{\text{focus}}$ do
                if $\neg s_{\text{property}}(v_{\text{focus}}) \in \sigma$ then
                    $R \leftarrow R \cup \{(s, \text{rsh:false}, v_{\text{focus}})\}$
                else if $s_{\text{property}}(v_{\text{focus}}) \notin \sigma$ then
                    $R \leftarrow R \cup \{(s, \text{rsh:unknown}, v_{\text{focus}})\}$
                else
                    $R \leftarrow R \cup \{(s, \text{rsh:true}, v_{\text{focus}})\}$
                end if
            end for
        else
            $V_{\text{value}} \leftarrow$ startRec($\left\{ v_{\text{focus}} \;\middle|\; \begin{array}{l} v_{\text{focus}} \in V_{\text{focus}} \\ \wedge (\sigma = \text{null} \vee \neg s_{\text{property}}(v_{\text{focus}}) \in \sigma) \end{array} \right\}, s_{\text{property}}$)
            $R \leftarrow$ validateNodesAgainstShape($G_D, G_S, V_{\text{value}}, s_{\text{property}}, \sigma$)
            endRec($V_{\text{value}}, s_{\text{property}}$)
        end if
    else
        for all $v_{\text{focus}} \in V_{\text{focus}}$ do
            $V_{\text{value}} \leftarrow$ getValueNodes($G_D, G_S, s, v_{\text{focus}}$)
            if $b$ then
                if $\exists_{v_{\text{value}} \in V_{\text{value}}} \neg s_{\text{property}}(v_{\text{value}}) \in \sigma$ then
                    $R \leftarrow R \cup \{(s, \text{rsh:false}, v_{\text{focus}})\}$
                else if $\exists_{v_{\text{value}} \in V_{\text{value}}} s_{\text{property}}(v_{\text{value}}) \notin \sigma$ then
                    $R \leftarrow R \cup \{(s, \text{rsh:unknown}, v_{\text{focus}})\}$
                else
                    $R \leftarrow R \cup \{(s, \text{rsh:true}, v_{\text{focus}})\}$
                end if
            else
                $V_{\text{value}} \leftarrow$ startRec($\left\{ v_{\text{value}} \;\middle|\; \begin{array}{l} v_{\text{value}} \in V_{\text{value}} \\ \wedge (\sigma = \text{null} \vee \neg s_{\text{property}}(v_{\text{value}}) \in \sigma) \end{array} \right\}, s_{\text{property}}$)
                $R \leftarrow R \cup$ validateNodesAgainstShape($G_D, G_S, V_{\text{value}}, s_{\text{property}}, \sigma$)
                endRec($V_{\text{value}}, s_{\text{property}}$)
            end if
        end for
    end if
    return $R$
end procedure
```

NodeConstraintComponent Restricts the set of value nodes to conform to the given node shape (value of mandatory parameter *sh:node*). If a value node does not conform to the node shape, there is a validation result with the value node as value. This is a referencing constraint component requiring conformance checking.

Algorithm 11 Validation of a set of focus nodes against a constraint of kind NodeConstraint-Component

```
procedure VALIDATENODESAGAINSTNODECONSTRAINT($G_D, G_S, V_{\text{focus}}, c, \sigma = \text{null}, b = \text{False}$)
    $(s, C, P) \leftarrow c$
    $s_{\text{node}} \leftarrow P(\text{sh:node})$
    $R \leftarrow \emptyset$
    for all $v_{\text{focus}} \in V_{\text{focus}}$ do
        $b_{\text{false}} \leftarrow \text{False}$
        $b_{\text{unknown}} \leftarrow \text{False}$
        for all $v_{\text{value}} \in$ GETVALUENODES($G_D, G_S, s, v_{\text{focus}}$) $\wedge \neg b_{\text{false}}$ do
            if $b$ then
                if $\neg s_{\text{node}}(v_{\text{value}}) \in \sigma$ then
                    $b_{\text{false}} \leftarrow \text{True}$
                else if $s_{\text{node}}(v_{\text{value}}) \notin \sigma$ then
                    $b_{\text{unknown}} \leftarrow \text{True}$
                end if
            else if $(\sigma \neq \text{null} \wedge \neg s_{\text{node}}(v_{\text{value}}) \in \sigma)$
                    $\vee (\sigma = \text{null} \wedge |$VALIDATENODESAGAINSTSHAPE($G_D, G_S, \{v_{\text{value}}\}, s_{\text{node}}$)$| > 0)$ then
                $R \leftarrow R \cup$ CREATERESULT($c, v_{\text{focus}}, v_{\text{value}}$, "does not conform to " $+ s_{\text{node}}$)
            end if
        end for
        if $b$ then
            if $b_{\text{false}}$ then
                $R \leftarrow R \cup \{(s, \text{rsh:false}, v_{\text{focus}})\}$
            else if $b_{\text{unknown}}$ then
                $R \leftarrow R \cup \{(s, \text{rsh:unknown}, v_{\text{focus}})\}$
            else
                $R \leftarrow R \cup \{(s, \text{rsh:true}, v_{\text{focus}})\}$
            end if
        end if
    end for
    return $R$
end procedure
```

AndConstraintComponent Restricts the set of value nodes to conform to all members of the given list of shapes (value of mandatory parameter *sh:and*). If a value node does conform to each member, there is a validation result with the value node as value. This is a referencing constraint component requiring conformance checking.

Algorithm 12 Validation of a set of focus nodes against a constraint of kind AndConstraintComponent

```
procedure VALIDATENODESAGAINSTANDCONSTRAINT($G_D, G_S, V_{\text{focus}}, c, \sigma = \text{null}, b = \text{False}$)
  $(s, C, P) \leftarrow c$
  $S_{\text{and}} \leftarrow P(\text{sh:and})$
  $R \leftarrow \emptyset$
  for all $v_{\text{focus}} \in V_{\text{focus}}$ do
    $b_{\text{false}} \leftarrow \text{False}$
    $b_{\text{unknown}} \leftarrow \text{False}$
    for all $v_{\text{value}} \in$ GETVALUENODES($G_D, G_S, s, v_{\text{focus}}$) $\wedge \neg b_{\text{false}}$ do
      if $b$ then
        if $\exists_{s_{\text{and}} \in S_{\text{and}}} \neg s_{\text{and}}(v_{\text{value}}) \in \sigma$ then
          $b_{\text{false}} \leftarrow \text{True}$
        else if $\exists_{s_{\text{and}} \in S_{\text{and}}} s_{\text{and}}(v_{\text{value}}) \notin \sigma$ then
          $b_{\text{unknown}} \leftarrow \text{True}$
        end if
      else if $(\sigma \neq \text{null} \wedge \exists_{s_{\text{and}} \in S_{\text{and}}} \neg s_{\text{and}}(v_{\text{value}}) \in \sigma)$
            $\vee (\sigma = \text{null} \wedge \exists_{s_{\text{and}} \in S_{\text{and}}} |$VALIDATENODESAGAINSTSHAPE($G_D, G_S, \{v_{\text{value}}\}, s_{\text{and}}$)$| > 0)$ then
        $R \leftarrow R \cup$ CREATERESULT($c, v_{\text{focus}}, v_{\text{value}}$, "does not conform to all members")
      end if
    end for
    if $b$ then
      if $b_{\text{false}}$ then
        $R \leftarrow R \cup \{(s, \text{rsh:false}, v_{\text{focus}})\}$
      else if $b_{\text{unknown}}$ then
        $R \leftarrow R \cup \{(s, \text{rsh:unknown}, v_{\text{focus}})\}$
      else
        $R \leftarrow R \cup \{(s, \text{rsh:true}, v_{\text{focus}})\}$
      end if
    end if
  end for
  return $R$
end procedure
```