UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Hasselt University
Martelarenlaan 42 |BE-3500 Hasselt

2024 | Faculty of Sciences

UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

VUB VRIJE UNIVERSITEIT BRUSSEL

Doctoral dissertation submitted to obtain the degrees of:
- Doctor of Sciences: Information technology | tUL
- Doctor of Sciences | VUB

Maxime Jakubowski

DOCTORAL DISSERTATION

Shapes Constraint Language: Formalization, Expressiveness, and Provenance

Promoters: Prof. Dr Jan Van den Bussche | UHasselt
Prof. Dr Bart Bogaerts | VUB

2024 | Faculty of Sciences

UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

VUB VRIJE UNIVERSITEIT BRUSSEL

Doctoral dissertation submitted to obtain the degrees of
- Doctor of Sciences: Information technology | tUL
- Doctor of Sciences | VUB

Maxime Jakubowski

DOCTORAL DISSERTATION

Shapes Constraint Language: Formalization, Expressiveness, and Provenance

DSI
DATA SCIENCE INSTITUTE
UHASSELT

Promoters: Prof. Dr Jan Van den Bussche | UHasselt
Prof. Dr Bart Bogaerts | VUB

D/2024/2451/48

Foreword

With this thesis, another chapter of my life comes to an end. Writing this foreword almost feels like writing a goodbye letter, and maybe that makes sense. During my PhD, and especially towards the end, I have felt immense gratitude. Never would I have thought that it would change me so fundamentally; or that I could do the research described in this thesis. None of this would have been possible if it wasn't for my two sharp, competent and caring advisors: Jan and Bart.

First and foremost, I would like to thank Jan. He guided me "day-to-day" and acted as my mentor. He trained me to be a researcher, taught me to be precise, and think critically. These skills are something I can carry with me throughout my life. Next to this, he showed me how to navigate conferences, work (scientifically) with others, and how to *properly* use LaTeX, among other things. Our good working relation contributed greatly to my motivation and output, and I am grateful to have worked with him.

From the beginning of my PhD, I also had the pleasure of working with Bart. Soon after starting, he became my second advisor. Our collaboration was very fruitful: the first two papers from this thesis [18, 20] exist because he pushed for them. Bart is quite creative, and has a talent for finding counterexamples to our claims which evidently greatly contributed to our work.

Both Jan and Bart had incredible patience with me, and both entertained and shot down many of my half thought-out ideas — always in a polite way. All of this is to say that I feel incredibly lucky to have had such a strong backing in doing this research.

Furthermore, I would like to thank the additional members of the jury: Dr Shqiponja Ahmetaj, Prof. Dr Anastasia Dimou, Prof. Dr Bas Ketsman, Dr Sławek Staworko, and Prof. Dr Stijn Vansummeren for taking the time to go go through this thesis and provide helpful suggestions which improved the overall quality. Also, an additional special thanks to Anastasia for laying the foundations of the ideas for Chapter 5.

Next, the positive environment at our group at UHasselt should not be taken for granted. The group grew dramatically since my time here, and I thank my colleagues with whom I spent many hours traveling, lunching, talking, and walking around the campus, and again had the patience to listen to me ramble about some topic I was occupied with at that moment.

Finally, I want to thank my extended family and friends, but especially my parents who, despite the difficult road, managed to raise my brother and me to have a good life. I'm thankful to my mother for helping me get through school, motivating me, and teaching me what it means to persevere. I know she would be proud. I'm thankful to my father, who supported me, gave me all the opportunities to go for higher education, and taught me how to care. Last but not least, I am grateful to Ellen, my partner, for reminding me I'm human and making our little apartment feel like home. And, not to forget, our dog Tofu who reminds me every day that I should also sometimes go touch some grass.

Nederlandstalige samenvatting

Het Resource Description Framework (RDF) is een standaard voor voor het representeren van kennis op het Web, gedefinieerd door het World Wide Web Consortium (W3C). Informatie wordt gerepresenteerd in een graaf-achtig datamodel gestructureerd als een verzameling subject-predikaat-object triples. Een verzameling van dergelijke triples wordt een *RDF graaf* genoemd. waarbij de triples gezien kunnen worden als bogen van subject naar object, met het predikaat als booglabel. Deze booglabels worden gebruikt om eigenschappen van knopen aan te geven.

Voor gebruikers van RDF grafen is het belangrijk om te weten welke eigenschappen te verwachten, of, meer algemeen, om te kunnen vertrouwen op bepaalde structurele beperkingen waaraan de graaf gegarandeerd voldoet. Daarvoor hebben we een declaratieve taal nodig om zulke beperkingen formeel te specificeren. In databasetermen: we hebben een schemataal nodig.

Shapes constraint language Het W3C beveelt Shapes Constraint Language (SHACL) aan als de schemataal voor RDF-grafen [86]. Het is de meest recente toevoeging aan de reeks technologieën die het beheer van RDF data vergemakkelijken. Anderen zijn SPARQL voor queries en OWL voor redeneren over RDF data.

Het belangrijkste doel van SHACL is het schrijven van een schema, in SHACL terminologie een *shapes graaf*, dat de verwachte structuur van een RDF graaf beschrijft. De RDF graaf waarvan een bepaalde structuur verwacht wordt een *data graaf* genoemd. Gegeven een shapes graaf en een data graaf, is de belangrijkste taak binnen SHACL om te controleren of de gegeven data graaf voldoet aan de vereisten gespecificeerd door de shapes graaf. Deze taak wordt conformiteitscontrole genoemd en wordt gedaan door software die een SHACL-validator wordt genoemd. Verder wordt vaak van validators verwacht dat ze niet alleen controleren op conformiteit, maar ook maar dat ze ook een validatierapport genereren waarin staat welke knopen welke delen van de shapes graaf schenden.

Een SHACL shapes graaf bestaat uit een verzameling *shapes* die structurele beperkingen op knopen zijn. Als we een shape evalueren op een knoop (we checken dan of de knoop voldoet aan de beschrijving van de shape), wordt die knoop een een *focusknoop* genoemd. Er zijn twee soorten shapes: node shapes en property shapes. Een node shape definieert direct beperkingen op de focusknoop, terwijl property shapes beperkingen definiëren op de *waardeknopen* van de focusknoop. De waardeknopen zijn de knopen die bereikbaar zijn via een eigenschap of paduitdrukking die gegeven wordt als een parameter voor de property shape (met behulp van het `sh:path` sleutelwoord in SHACL).

Als eenvoudig voorbeeld zullen we enkele shapes beschouwen in een access control setting. Basis gebruikers kunnen resources openen en aanmaken, terwijl gemachtigde gebruikers ook nieuwe gebruikers kunnen toevoegen aan het systeem en hen goedkeuren om ook gemachtigde gebruikers worden. Beschouw de volgende data graaf:

```
:admin_user a :Admin ;
  :adds :user_a ;
  :adds :user_b ;
  :approves :user_b .

:user_b :adds :user_c ;
  :approves :user_c .

:user_a :accesses :resource1 .
:user_b :accesses :resource1 .
:user_c :accesses :resource1 ;
  :creates :resource2 .
```

Het idee achter de eerste paar lijnen van de bovenstaande beschrijving is dat de `:admin_user` gebruiker de andere gebruikers `:user_a` en `:user_b` toevoegt aan het systeem. De `:admin_user` gebruiker machtigt verder ook gebruiker `:user_b`, wat betekent dat `:user_b` nu ook gemachtigd is binnen dit systeem.

Beschouw de volgende node shape die bestaat uit een knooptest en twee property shapes:

```
:basicUserShape a sh:NodeShape ;
  sh:nodeKind sh:IRI ;
  sh:property [
    sh:path :accesses ;
    sh:minCount 1
  ] ;
  sh:property [
    sh:path :creates ;
    sh:maxCount 0
  ] .
```

De shape `:basicUserShape` specifieert drie vereisten voor een gegeven focusknoop. Ten eerste moet het focusknoop een IRI zijn. Het sleutelwoord `sh:property` geeft aan dat de focusknoop ook moet moet voldoen aan een property shape. De eerste property shape gaat over de waardeknopen die bereikbaar zijn met de `:accesses` eigenschap. Specifiek moet er ten minste één zo'n waardeknoop zijn. De tweede property shape gaat over de waardeknopen die bereikbaar zijn met de eigenschap `:creates`. In dit geval kunnen er maximaal 0 zijn. In natuurlijke taal zegt deze shape: "De focusknoop is een IRI en `:accesses` minstens één resource, maar maakt er geen nieuwe aan." Dus in onze voorbeeldgraaf voldoen enkel `:user_a` en `:user_b` aan `:basicUserShape`.

Een shape op zich is geen beperking op een graaf, maar op een knoop. Om een beperking over een hele graaf te realiseren, voegen we een *target declaration* toe aan de shape definitie. Een target declaration is simpelweg een (eenvoudige) query die voor een gegeven shape bepaalt wat precies de focusknopen zijn. Bijvoorbeeld, we zouden een target declaration kunnen toevoegen aan het bovenstaande dat zegt dat elke knoop die voorkomt als een subject in een triple met als predikaat `:approves` moet voldoen aan `:basicUserShape`. In dat geval voldoet onze data graaf aan de beschreven beperking.

De semantiek van SHACL is nodeloos ingewikkeld als we deze bestuderen vanuit de oorspronkelijke syntax. Daarom is het tweede hoofdstuk van de thesis gewijd aan het formaliseren van SHACL op een manier die het geschikt maakt voor verschillende onderzoeken.

Expressiveness Gezien de definitie van SHACL als een beperkingstaal, zijn gebruikers vaak geïnteresseerd in wat je er eigenlijk mee kunt doen; welke shapes kun je eigenlijk schrijven? Wanneer een taal wordt gedefinieerd, worden veel criteria opgesteld over wat de taal zou moeten kunnen beschrijven. Vaak worden deze criteria dan direct in de taal opgenomen. Een eenvoudig voorbeeld zijn de klassenbeperkingen in SHACL. Hiermee kunnen shapes geschreven worden die aangeven dat de focusknoop van een bepaalde RDF klasse (of subklasse daarvan) moet zijn. Het is gemakkelijk om te zien dat deze functie technisch gezien overbodig is: als we deze restrictiecomponent uit SHACL zouden verwijderen, zou de uitdrukkingskracht hetzelfde blijven, omdat we het al kunnen uitdrukken met andere restrictiecomponenten. Bijvoorbeeld, gegeven een IRI `<c>` elke shape van de vorm:

```
[ a sh:NodeShape ;
  sh:class <c>
]
```

kan geschreven worden als de shape:

```
[ a sh:PropertyShape ;
  sh:path ( rdf:type [ sh:zeroOrMorePath rdfs:subClassOf ] ) ;
  sh:hasValue <c>
]
```

Het is duidelijk dat beperkingen over klassen nuttig zijn, dus is het opgenomen in de taal. Ons voorbeeld laat echter zien dat de complexe pad expressies fundamenteler zijn voor SHACL. We hebben deze, dus we kunnen al klassebeperkingen beschrijven.

Deze bespreking van de klassenbeperkingen is vrij duidelijk; het is gemakkelijk in te zien dat het uit te drukken is door andere componenten van SHACL te gebruiken. Nu rijst de natuurlijke vraag op: gebeurt dit ook op andere plaatsen in SHACL? Kunnen we bijvoorbeeld het Equals Constraint Component uitdrukken met andere SHACL features? Het is niet duidelijk of dit mogelijk is, maar hoe kun je er zeker van zijn dat het niet uitdrukbaar is? Dit zijn de expressiviteitsvragen die we onderzoeken in het derde hoofdstuk (en om de vraag te beantwoorden, nee, je kan het equals component niet met andere componenten uitdrukken).

Vragen over expressiviteit komen ook naar voren in ander onderzoek naar SHACL. Onlangs onderzochten Ferranti et al. [40] het uitdrukken van de integriteitsbeperkingen van Wikidata in SHACL, waaruit bleek dat SHACL-core mogelijk niet in staat is om sommige van die beperkingen uit te drukken.

Recursion Van een shape wordt gezegd dat het *recursief* is als de definitie terugverwijst naar zichzelf. De W3C laat de definitie voor recursieve shapes open, en dit motiveert ons om mogelijke semantiek voor dergelijke shapes te onderzoeken. De literatuur beschrijft verschillende recursieve semantieken voor SHACL, maar er is ook veel werk gedaan aan recursieve semantiek voor andere logische formalismen. We passen een algebraïsch raamwerk toe, Approximation Fixpoint Theory, om een sterke formele basis te leggen voor de studie van recursieve semantiek voor SHACL. We vergelijken de resulterende semantiek met die uit de literatuur.

Provenance Het bovenstaande ging vooral over fundamentele aspecten van SHACL. Het is ook interessant om alternatieve semantieken voor SHACL te verkennen, zoals “provenance” semantiek.

Gegeven een SHACL shapes graaf en een data graaf die voldoet aan die shapes graaf, willen we weten welke subset van de data graaf *relevant* is om te beslissen dat het valideert. Deze subset van de data graaf wordt de data provenance genoemd.

Er zijn veel intuïties voor welke gegevens relevant zijn. De provenance zou gezien kunnen worden als de subset van de graaf “uitgetraceerd” door de shape door de waarden van `sh:path` te volgen in de property shapes. Een andere intuïtie is dat we graag een subset van de graaf die de validator "bekijkt" tijdens het valideren van de gegevens. Deze begrippen zijn nuttig, maar onnauwkeurig. Onze benadering is om in ieder geval rekening te houden rekening te houden met de zogeheten *sufficiency* eigenschap van de provenance. De sufficiëntie-eigenschap stelt, informeel, dat de resulterende provenance nog steeds voldoet aan de beperkingen gegeven door de shapes graph. Sufficiency vertelt ons dat onze provenance eigenlijk relevant is op een heel precieze manier — het is duidelijk dat gegevens die bijdragen aan conformiteit relevant moeten zijn.

We definiëren de provenance eerst op shape-niveau (in tegenstelling tot shape graaf niveau). Dus, gegeven is een data graaf G, een shape s en een knoop n uit G die voldoet aan s, definiëren we de subset van de data graaf die we de *neighborhood* noemen. Deze definitie kan later dan ook veralgemeend worden naar shapes graphs.

Neighborhoods kunnen duidelijk worden gezien als een aanvullende semantiek voor SHACL die een opvraagmechanisme definieert. We noemen deze opvraagsemantiek *shape fragements*.

Gegeven een shape en een data graaf, is het shape fragment eenvoudigweg de neighborhood van alle knopen die voldoen aan die shape. Als er target declaraties zijn, beschouwen we alleen de knooppunten die worden bepaald door de target declaration (en nemen we de informatie op die hen target). Het kan op dezelfde manier gedefinieerd worden voor verzamelingen van shapes of voor shape grafen.

Men kan zich het nut van zo'n terughaalmechanisme wel voorstellen. Wanneer we grote RDF grafen beschouwen, kan het zijn dat een shape graaf slechts een deel beschrijft dat relevant is voor het beoogde gebruik. Het ophalen van het shape fragment van de data graaf geeft ons dan een (mogelijk) kleinere RDF graaf die gemakkelijker te verwerken is en nog steeds voldoet aan de beperkingen en dus toch relevante gegevens bevat.

Abstract

The Shapes Constraint Language (SHACL) is a W3C-proposed schema language for expressing structural constraints on RDF graphs. Constraints on nodes are called “shapes”, and when shapes are coupled with so-called “target declarations”, specifying which nodes need to adhere to which shapes, we have a complete constraint on RDF graphs. We study several aspects of this language. First, recent formalizations show a striking resemblance with description logics. We build on top of these formalizations to come to an understanding of SHACL as a logic. Furthermore, because the SHACL specification only defines semantics for *non-recursive* SHACL, some efforts have been made to allow *recursive* SHACL schemas. We argue that for defining and studying semantics of recursive SHACL, lessons can be learned from research in non-monotonic reasoning. We look at the proposed semantics from the literature and compare it with techniques from well-established research from non-monotonic reasoning.

Next, SHACL expressions can use three fundamental features that are not so common in similar logics. These features are equality tests; disjointness tests; and closure constraints. It is not clear how the presence of these non-standard features impacts the *expressiveness* of SHACL. We show that each of the three features is primitive: using the feature, one can express boolean queries that are not expressible without using the feature. We also show that the restriction that SHACL imposes on allowed targets is inessential, as long as closure constraints are not used. In addition, we show that enriching SHACL with “full” versions of equality tests, or disjointness tests, results in a strictly more powerful language.

Lastly, we propose provenance semantics for SHACL. We propose the notion of *neighborhood* of a node v satisfying a given shape in a graph G. This neighborhood is a subgraph of G, and provides data provenance of v for the given shape. We establish a correctness property for the obtained provenance mechanism, by proving that neighborhoods adhere to the Sufficiency requirement articulated for provenance semantics for database queries. As an additional benefit, neighborhoods allow a novel use of shapes: the extraction of a subgraph from an RDF graph, the so-called shape fragment. We compare shape fragments with SPARQL queries. We discuss implementation strategies for computing neighborhoods, and present initial experiments demonstrating that our ideas are feasible.

Contents

1 Introduction

The Resource Description Framework (RDF) is a standard for representing knowledge on the Web recommended by the World Wide Web Consortium (W3C) [78]. Information is represented in a graph-like data model structured as a set of subject-predicate-object triples. A set of such triples is referred to as an *RDF graph* where the triples can be seen as edges from subject to object, with the predicate as edge-label. These edge-labels are used to indicate properties of nodes.

For consumers of RDF graphs, it is important to know what properties to expect, or, more generally, to be able to rely on certain structural constraints that the graph is guaranteed to satisfy. Therefore, we need a declarative language to specify such constraints formally. In database terms, we need a schema language.

The W3C recommends the Shapes Constraint Language (SHACL) [86] as the schema language for RDF graphs. It is the most recent addition to the set of technologies facilitating RDF data management, others including SPARQL for querying, and OWL for reasoning.

In this thesis, we study several aspects of SHACL. This chapter starts with a short introduction into the topics of this thesis, starting with a short introduction to SHACL.

Throughout this chapter we will use Turtle notation [81] for RDF, and we assume the following prefixes are defined:

```
@prefix : <https://www.mjakubowski.info/vocabulary/>
@prefix sh: <http://www.w3.org/ns/shacl#>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

1.1 SHACL in a nutshell

The main purpose of SHACL is to write a schema, in SHACL terminology called a *shapes graph*[1], that describes the expected structure of an RDF graph. The RDF graph of which some structure is expected is referred to as the *data graph.* Given a shapes graph and a data graph, the main task is to check whether the given data graph satisfies the requirements specified by the shapes graph. This task is called *conformance checking* and it is done by software called a *(SHACL) validator.* Furthermore, it is often expected of validators to not only check for conformance, but to also generate a *validation report* specifying which nodes violate which parts of the shapes graph.

[1] The reason this is called a “graph” is that the SHACL syntax is written in RDF.

Shapes

A SHACL shapes graph consists of a set of *shapes* which are structural constraints on nodes. Shapes have a name which is a blank node or an IRI. When we evaluate a shape on a node, that node is called a *focus-node*. There are two types of shapes: node shapes and property shapes. A node shape directly defines constraints on the focus node, while property shapes defines constraints on the *value nodes* of the focus node. The value nodes are the nodes reachable through a property or path expression that is given as a parameter to the property shape (using the `sh:path` keyword in SHACL).

Throughout this chapter, we will use examples from the access control setting. For this section, we will simply define some shapes about users in this setting. Basic users can access and create resources, while power users can also add users to the system and approve of them to also become power users. Consider the following data graph:

```
:admin_user a :Admin ;
  :adds :user_a ;
  :adds :user_b ;
  :approves :user_b .

:user_b :adds :user_c ;
  :approves :user_c .

:user_a :accesses :resource1 .
:user_b :accesses :resource1 .
:user_c :accesses :resource1 ;
  :creates :resource2 .
```

The idea of the first few lines is that the `:admin_user` adds users `:user_a` and `:user_b` to the system. They also approve of `:user_b`, meaning that `:user_b` is now also trusted within the system.

Consider the following node shape that consists of one node test and two property shapes:

```
:basicUserShape a sh:NodeShape ;
  sh:nodeKind sh:IRI ;
  sh:property [
    sh:path :accesses ;
    sh:minCount 1
  ] ;
  sh:property [
    sh:path :creates ;
    sh:maxCount 0
  ] .
```

The shape name is `:basicUserShape`. It specifies three requirements for a given focus node. First, the focus node should be an IRI, as opposed to a blank node or a literal. The `sh:property` keyword indicates that the focus node must also adhere to a property shape. The first property shape is about the value nodes reachable with the `:accesses` property. Specifically, there must be at least one such value node. The second property shape is about the value nodes reachable with the `:creates` property. In this case, there may be at most 0. In natural language, this shape states: “The focus node is an IRI and `:accesses` at least one resource, but `:creates` none.” So, in our example graph, only `:user_a` and `:user_b` satisfy `:basicUserShape`.

SHACL has many different features for writing constraints. These features are called *constraint components*. The previous example made use of a *value type constraint component*, here indicated with the `sh:nodeKind` keyword, to check whether the focus node is an IRI, but also of *cardinality constraint components* to count the number of value nodes, indicated with the `sh:minCount` and `sh:maxCount` keywords. SHACL has many constraint components and, in this thesis, we will restrict ourselves to the *core* constraint components[2]. Next follows some examples of some of the components that illustrate the capabilities of SHACL.

Property Pair Constraint Components. In property shapes, we can compare the value nodes to another set of nodes (reachable by some other property) in some predefined ways. The most important are *equality* and *disjointness* checks between the two sets. Consider the following node shape:

```
:powerUserShape a sh:NodeShape ;
    sh:not [ a sh:PropertyShape ;
             sh:path :adds ;
             sh:disjoint :approves ] .
```

This shape asks of the focus node that the set of value nodes, i.e., the nodes reachable with the `:adds` property, is *not* disjoint from the set of nodes reachable with the `:approves` property. In natural language: "The focus node approves at least one user they also added." So, in our graph `:admin_user` and `:user_b` satisfy `:powerUserShape`.

Shape-based Constraint Components. We can refer to other shapes as well. Most notably, we can combine this with the counting from the cardinality constraint components to count only value nodes that conform to some other given shape. For example, consider the shape:

```
:authorizedUserShape a sh:PropertyShape ;
    sh:path [ sh:zeroOrMore [ sh:inverseProperty :approves ] ] ;
    sh:qualifiedValueShape :isAdminShape ;
    sh:qualifiedMinCount 1 ;
  ] .

:isAdminShape a sh:PropertyShape ;
  sh:path rdf:type ;
  sh:hasValue :Admin .
```

A first thing to note is that this shape makes use of a (complex) path expression. SHACL supports path expressions similar to those of SPARQL. This shape states that there must be at least one value node reachable, by following inverse `:approves` properties, that conforms to `:isAdminShape`. The keyword `sh:qualifiedValueShape` is used to refer to a shape, together with the `sh:qualifiedMinCount` to denote the desired cardinality. In our example graph, `:admin_user`, `:user_b`, and `:user_c` satisfy `:authorizedUserShape`. Note that it is different from writing:

```
:altAuthorizedUserShape a sh:PropertyShape ;
    sh:path [ sh:zeroOrMore [ sh:inverseProperty :approves ] ] ;
    sh:node :isAdminShape ;
    sh:minCount 1 ;
  ] .
```

where `sh:node` is used to refer to another shape. This seems to express the same shape, however, all constraints used in a property shape apply to all value nodes, i.e., it is an implicit universal quantifier. When a cardinality constraint is then used, it is separate

[2] `https://www.w3.org/TR/shacl/#core-components`

from the other constraints, and only refers the number of value nodes. In words, the `:altAuthorizedUserShape` states that all value nodes conform to `:isAdminShape` and there is at least one node in the set of value nodes. In our example graph, only `:admin_user` satisfies `:altAuthorizedUserShape`.

Closed Constraint Component. Finally, another interesting feature is *closedness*. It states that only a select few properties are allowed for a focus node. These select few properties can be given explicitly by the `sh:ignoredProperties` keyword, or are implied by the structure of the shape: the set of non-blank nodes obtained by following the `sh:property` keyword, followed by the `sh:path` keyword. Consider `:basicUserShape` defined above. We can expand the shape by adding the following triples:

```
:basicUserShape sh:closed true ;
  sh:ignoredProperties ( rdf:type ) .
```

The shape is *closed*, meaning the only allowed properties are give by `sh:ignoredProperties`, i.e., `rdf:type`, but we also allow the properties related to the property shapes: `:accesses` and `:creates`. In natural language: “The focus node has no other properties than `rdf:type`, `:accesses`, or `:creates`; and must have an `:accesses` property and no `:creates` property.” So in our example graph, only `:user_a` now satisfies `:basicUserShape`.

Clearly, there are many intricacies in the semantics of SHACL that make it unsuited to study directly. Thereto, Chapter 2 is dedicated to formalizing SHACL in such a way that makes it fit for the different investigations performed in this thesis. There are many other constraint components than those demonstrated here. However, the ones discussed here are illustrative of the core SHACL features. Detailing the exact semantics of all core SHACL features is done in Chapter 2, Section 2.4.1.

Targeting

A set of shape definitions on its own does not allow us to validate a graph in SHACL. Every shape can have a *target declaration* associated with it. A target declaration is a query that determines all the focus nodes for a given shape. There are five types of target declarations allowed in SHACL. Most of them are parameterized with an IRI or blank node t:

Node targets. There is one focus node, which is the parameter t (regardless whether t occurs in the graph.)

Class-based targets. Targets all nodes that are of RDF class t.

Implicit class targets. Targets all nodes whose RDF class is the shape name.

Subjects-of targets. Targets all nodes that are the subject of a triple where the predicate is t.

Objects-of targets. Targets all nodes that are the objects of a triple where the predicate is t.

We can add target declarations to the shapes from the previous section to make a complete shapes graph. Adding the Subjects-of target triple

```
:authorizedUserShape :targetSubjectsOf :approves
```

to the shapes graph gives us a complete constraint on the graph. In this case, our example data graph conforms because all nodes that approve another node are indeed authorized users. However, if we add the triple `:user_a :approves :user_d` to the data graph, it would no longer conform, as `:user_a` does not satisfy `:authorizedUserShape`.

1.2 Expressiveness

Given the definition of SHACL as a constraint language, users are often interested in what you can actually *do* with it; what shapes can you actually write? When a language is defined, many criteria are postulated on what the language should be able to describe. Often, these are then *directly* included in the language. A simple example is the *class constraint component* in SHACL. This allows for writing shapes stating that the focus node must be of a certain RDF class (or subclass thereof). It is easy to observe this feature is technically redundant: if we would remove this constraint component from SHACL, the expressive power would remain the same, because we can already express it using other constraint components. For example, given an IRI `<c>` every shape of the form:

```
[ a sh:NodeShape ;
  sh:class <c>
]
```

can be written as the shape:

```
[ a sh:PropertyShape ;
  sh:path ( [ sh:zeroOrMorePath rdfs:subClassOf ] rdf:type ) ;
  sh:hasValue <c>
]
```

Clearly, constraints about classes are useful, so it is included in the language. However, our example shows that the complex path expressions are more fundamental to SHACL. We have these, so we can already do class constraints.

This discussion of the class constraint component is quite clear cut; it is easy to see that it is expressible by other features. Now, a natural question arises: does this also happen for other constraint components? Can we, for example, express the *equals constraint component* using other SHACL features? In this case it is not clear whether this is possible, but how can you be certain that it is *not* expressible? These are the expressiveness questions we investigate in this thesis (and to actually answer the question, no, you cannot express equality with the other features).

To add to the relevance of this kind of investigation, the SHACL community is interested in *extending* SHACL with other constraint components called the *DASH Constraint Components* [56]. It turns out that most of these constraint components are not easily (or at all) expressible with SHACL as it is, but if we extend SHACL with a more powerful version of the equality constraint component, the DASH extensions become expressible![3] As a concrete example, DASH proposes a constraint *dash:nonRecursive* on property shapes that states that there must not be a path (from `sh:path`) from the focus node, to the focus node. In other words, there may not be a self-loop (following the path) in the data graph for the focus node. This can be expressed with the shape:

```
:noLoop a sh:NodeShape
  sh:not [ a sh:PropertyShape ;
           sh:path <path> ;
           sh:equals [ sh:zeroOrOnePath <path> ]
         ] .
```

However, this shape is technically not a correct SHACL shape because the `sh:equals` keyword may only have IRIs as their value, not blank nodes (and thus not paths). Nevertheless, the meaning is clear.

The point here is not to say that DASH is superfluous, rather, it is simply to give insight in what is fundamentally happening when you add features to SHACL. It turns out that adding

[3] This has been described in this blog post: `https://mjakubowski.info/posts/datashapes`

full-path support to the equality feature lets us also express some other DASH constraints, indicating that adding full equality to SHACL is useful. Furthermore, it may give insights to developers implementing SHACL validators: not every added feature requires implementing new algorithms to support that feature.

Lastly, questions of expressiveness also appear in other research on SHACL. Recently, Ferranti et al. [40] investigated expressing the integrity constraints of Wikidata in SHACL, indicating SHACL-core may not be able to express some of the constraints.

1.3 Semantics for recursion

A shape is said to be *recursive* if its definition refers back to itself. The W3C leaves recursive shapes undefined and, in this section, we will explore possible semantics for such shapes. Different semantics from the literature will be discussed [6, 18, 28, 30], without defining them formally. Formal definitions will be given and explored in Chapter 3. The proposed approaches to recursion are directly inspired from the logic programming literature and are called: Supported Model Semantics, Well-Founded Semantics, Stable Model Semantics, and Stratified Recursion [42, 93, 97].

Setting the stage

The running example will be the access control setting from the previous section, where users can be authorized to access resources or not. The data graph will represent an overview of an access control scheme where there are users and resources. Users can approve other users to access files. There is a distinguished user, the administrator, who is authorized by default.

A SHACL shape graph describes what is a valid access control scheme, i.e., if a data graph conforms to the SHACL shapes graph, it is a valid access control scheme.

Positive recursion A shape is said to be recursive if it references itself (directly, or indirectly). Consider the following access control shapes graph:

```
:authorized a sh:NodeShape ;
  sh:or (
    [ a sh:NodeShape ;
      sh:hasValue :admin_user
    ] ;
    [ a sh:PropertyShape ;
      sh:path [ sh:inversePath :approves ] ;
      sh:qualifiedValueShape :authorized ;
      sh:qualifiedMinCount 1
    ]
  ) .

:authorized sh:targetSubjectsOf :accesses .
```

In natural language, the `:admin_user` node is `:authorized` and every node that is approved by an `:authorized` node is also `:authorized`. We target all nodes that access something. In essence, this shapes graph validates when all nodes that access a resource are `:authorized`.

There are two things to note about this shape. First, it is clearly recursive, as it mentions itself through the `sh:qualifiedValueShape`. Second, it mentions itself *positively*, that is, the recursion does not happen through some kind of negation (like `sh:not`).

When talking about recursion, negation adds an extra layer of complexity to the discussion. Almost all the differences between the proposed semantics are due to negation.

Remark 1.1. The `:authorized` shape is very similar to `:authorizedUserShape` from the previous section, but uses recursion. Indeed, also `:authorized` can be written without recursion. However, it is possible to adjust `:authorized` in such a way that it is not easily (or at all) expressible without recursion. For example, adding the triple (`:authorized`, `sh:class`, `:person`) to the shape adds the constraint that every node on the path of inverse `:approves` properties must be of RDF class `:person`.

Semantics for positive recursion We will start exploring the meaning of our `:authorized` shape. Consider the following data graph:

```
:admin_user :approves :user_a .
:user_b :approves :user_c .
:user_c :approves :user_b .
```

Let us focus first on deciding which users could actually be assigned the `:authorized` shape, forgetting for a moment the targeting specified in our shapes graph. Which of the users could be said to be `:authorized`? That is a question we can only answer when we have defined recursive semantics for SHACL.

Stratified recursion, Well-Founded, and Stable Model semantics all agree that the set of `:authorized` nodes needs to be only `:admin_user` and `:user_a`, as `:admin_user` is authorized by default and approves of `:user_a`. Supported Model semantics, however, states that there is an *additional* possible set of `:authorized` nodes: all of them!

The idea behind Supported Model semantics is that there exists a possible assignment of shape names to nodes, such that all assigned nodes satisfy the shape. In this case, when both `:user_b` and `:user_c` are assigned to be `:authorized`, then they also both satisfy the `:authorized` shape. Indeed, `:user_b` is approved by an authorized user (`:user_c`) and similarly for `:user_b`. Clearly, this does not match our intention, since this would allow every user to authorize themselves!

For the other semantics, the idea in this case is to have a minimal assignment of shape names to nodes. This is commonly referred to as the *minimal model semantics* in logic programming literature.

To complete our discussion on positive recursion, we will address the targeting mechanism of SHACL. Consider the following addition to the data graph:

```
:user_b :accesses :resource1 .
```

Now, `:user_b` is also targeted, and thus this data graph does not validate under Stratified, Stable Model, and Well-Founded semantics. Arguably, no reasonable semantics would ever assign `:authorized` to `:user_b`.

The Supported Model semantics has two possible assignments, but which one do we consider? There are two ways to look at this, according to the literature. First, *brave validation* states that we will just require there to be just *one* assignment that assigns `:user_b` to be `:authorized`. The other, *cautious validation* states that we require *all* possible assignments to assign `:user_b` to be `:authorized`. Thus, to make this concrete, if we use Supported Model semantics with cautious validation it will agree with the other semantics and the data graph will not validate. Otherwise, with brave validation, Supported Model semantics dictates that the data graph validates.

Adding negation

Consider the `:potentialThreat` shape defined in Listing 1.1. In natural language, this shape states that a node is a `:potentialThreat` if it is not `:authorized` and accesses a resource. All nodes that access a resource and are not `:authorized` are a `:potentialThreat`. We want to ensure that all users that are a `:BannedUser` are a `:potentialThreat`.

Listing 1.1: The :potentialThreat shape.

```
:potentialThreat a sh:NodeShape ;
  sh:property [
    sh:path :accesses ;
    sh:minCount 1
    ] ;
  sh:not :authorized .
:potentialThreat sh:targetClass :BannedUser .
```

Now, consider the following data graph:

```
:admin_user :approves :user_a .
:user_b :approves :user_c .
:user_c :approves :user_b .

:user_b :accesses :resource1 .
:user_c :accesses :resource1 .

:user_b a :BannedUser .
```

Which users would be assigned to be a :potentialThreat, and which ones would be assigned to be :authorized? The idea of minimal assignments is not going to help us here. All users will be assigned a label. They are either :authorized or a :potentialThreat.

Of course, the intuition is that :user_b and :user_c do not have any grounded reason to be :authorized. They are only so in a self-supporting way. So, we would likely want them to be a :potentialThreat.

One idea is to first resolve the *positive* recursion part of the shapes graph. So we would apply minimal models on :authorized and afterward try to assign :potentialThreat. This is exactly the idea behind Stratified Recursion. This only works when we look at the shapes graph in "layers" where we first resolve the positive recursion and work our way up. Again, Stable Model and Well-Founded semantics agree with this approach. Only Supported Model semantics, like before, considers two possible assignments. The one where all users are :authorized and the one where :user_b and :user_c are a :potentialThreat.

An interesting remark here is that with Supported Model semantics neither brave nor cautious validation will always result in a satisfying validation result. Indeed, with cautious validation :user_b will not be marked a :potentialThreat because there is an assignment where :user_b is :authorized. With brave validation, we can do whatever we want. If :user_b is not banned, it would be fine, because there is an assignment where :user_b is :authorized!

We can not always stratify easily

It is not always possible to look at the validation process in a stratified manner. Consider a new shapes graph redefining the :authorized shape, given in Listing 1.2. This shape is very similar to the :authorized shape from before, only now, you can only be :authorized if you are also never reported by an :authorized user. Note that this shape cannot be stratified as we did with the :potentialThreat before. It directly references itself negatively. We will next look at the following data graph scenario from Listing 1.3. It is easy to see that all these users would be :authorized. There is nothing special going on as long as we do not have anyone reporting anyone. We will add the following triple: (:user_b, :reports, :user_c) Because :user_c has been reported, they are no longer :authorized and neither is

Listing 1.2: The non-stratified `:authorized` shape definition.

```
:authorized a sh:NodeShape ;
  sh:or (
    [ a sh:NodeShape ;
      sh:hasValue :admin_user
    ]
    [ sh:and (
        [ a sh:PropertyShape ;
          sh:path [ sh:inversePath :approves ] ;
          sh:qualifiedValueShape :authorized ;
          sh:qualifiedMinCount 1
        ]
        [ sh:not [ a sh:PropertyShape ;
               sh:path [ sh:inversePath :reports ] ;
               sh:qualifiedValueShape :authorized ;
               sh:qualifiedMinCount 1
                 ]
        ]
     )
    ]
  ) .
:authorized sh:targetSubjectsOf :accesses .
```

Listing 1.3: A possible data graph scenario for the non-stratified setting.

```
:admin_user :approves :user_a ;
  :approves :user_b .
:user_a :approves :user_c .
:user_c :approves :user_d .
```

`:user_d`, even though `:user_d` has not been explicitly reported. All this the semantics agree on. However, we can add the following triples to the data graph:

```
:user_a :reports :user_b .
:user_b :reports :user_a .
```

Now, it is unclear what the assignments should be exactly.

The Stable Model semantics will tell you there are two possible assignments: one where `:user_a` is `:authorized` and `:user_b` is not, and one where the opposite is true. Again, to know how to choose between these two assignments, we can apply the ideas of cautions and brave validation.

The Well-Founded semantics will take a different approach. Unlike all the semantics discussed before, it will not just assign shapes to nodes: in the Well-Founded semantics, assignments are *three-valued*. Assignments are true, false, or unknown. In this example, it will say the assignment of `:authorized` for `:user_a` and `:user_b` are unknown.

Short summary

First, Stratified Recursion is the only approach that somehow restricts the kinds of recursion you can write. The other approaches are also nicely defined on all possible shapes graphs. Second, Stable Model and Supported Model semantics consider multiple possible assignments. In these two semantics, it is useful to then distinguish cautious and brave validation. Third, the Well-Founded semantics considers three-valued assignments, but only one assignment is defined for a shapes graph, disregarding the need for the cautious and brave distinction. Finally, all semantics, except for Supported Model semantics, agree if we only consider shape graphs with only positive recursion — even on Stratified shape graphs!

1.4 Provenance and SHACL

Until now, we have primarily discussed the fundamental semantics for SHACL. In this section, we will explore another useful semantics: data provenance for SHACL.

Given a SHACL shapes graph and a data graph that conforms to that shapes graph, we want to know what subset of the data graph is *relevant* to decide that it conforms. This subset of the data is called the *data provenance*.

There are many intuitions of what data is relevant. The provenance could be seen as the subset of the graph “traced out” by the shape by following the values of `sh:path` in the property shapes. Another intuition is that we would like to have a subset of the graph that the validator “looked at” while validating the data. These notions are useful but imprecise, as will be demonstrated later in this section. Our approach is to at least take into consideration the so-called *sufficiency* property of the provenance. The sufficiency property states, informally, that the resulting provenance still conforms to the shapes graph. Or, at the level of shapes (without targets), all nodes from the input data graph still satisfies the same shapes in the provenance. Sufficiency tells us that our provenance is actually relevant in a very precise way — clearly, data that contributes to conformance must be relevant.

Our discussion will focus on the provenance of a shape without target declarations. So, given a data graph G, a shape s, and a node n from G that conforms to s, we will define the provenance which we call the *neighborhood* of n in G according to s. This can then later be generalized for shapes graphs (including target declarations) as opposed to just shapes.

Neighborhoods by example

We will use the following data graph for our examples:

```
:user_a a :Admin;
  :accesses :resource1 .

:user_b a :Admin;
  :accesses :resource1 ;
  :accesses :resource2 .

:user_c a :User;
  :accesses :resource1 ;
  :accesses :resource2 .

:resource1 a :Resource .
:resource2 a :Resource .
```

Suppose we have a shape defining a standard resource as one that is accessed by at least one user (or admin):

```
:standardResource a sh:NodeShape ;
  sh:class :Resource ;
  sh:property [
    sh:path [ sh:inversePath :accesses ] ;
    sh:minCount 1 ;
  ] .
```

Clearly, both `:resource1` and `:resource2` satisfy the shape. We will look at the neighborhood of `:resource1`. Looking at the intuition behind neighborhoods, the triple stating that `:resource1` is a `:Resource` is definitely in the neighborhood, as well as *at least one* of the triples stating that it is accessed by a user. However, there is no order on the triples defined in RDF, so picking any of the three triples is somewhat arbitrary. We could define provenance to be non-deterministic, and say that there are three neighborhoods. However, our approach is to have a *deterministic* provenance definition and thus, we choose to define the neighborhood as:

```
:resource1 a :Resource .
:user_a :accesses :resource1 .
:user_b :accesses :resource1 .
:user_c :accesses :resource1 .
```

Note that this does not follow the intuition that we want to have the subgraph that the validator looks at. Any reasonable validator would only need to consider one `:accesses` triple to know that `:resource1` conforms to `:standardResource`.

We will continue with another example. Consider the shape defining unpopular resources as resources that are accessed by at most two users:

```
:unpopularResource a sh:NodeShape ;
  sh:class :resource ;
  sh:property [
    sh:path [ sh:inversePath :accesses ] ;
    sh:maxCount 2 ;
  ] .
```

Only `:resource2` satisfies this shape. There are multiple options again for what the neighborhood could be. There are two deterministic options. Either the neighborhood contains all `:accesses` triples or none! According to our intuition of wanting to "trace out" the shape the former is the suiting definition. However, our definition will choose to not include any :accesses triples in this case. Our design philosophy here is that neighborhoods should be somehow *minimal.* Choosing not to include any :accesses triples is minimal, and still makes :resource2 conform to the shape. Although this may be counterintuitive, this design choice makes sense in light of sufficiency. We will demonstrate a similar situation with a slightly more complex shape containing the same ideas:

```
:specialResource a sh:NodeShape ;
  sh:class :resource ;
  sh:property [
    sh:path [ sh:inversePath :accesses ]

    sh:minCount 1 ;

    sh:qualifiedValueShape [ sh:not [ sh:class :admin ]] ;
    sh:qualifiedMaxCount 1 ;
  ]
```

This shape consists out of three constraints. First, the node must be a resource. Second, the resource must be accessed by at least one user. Finally, at most one user that accesses this resource may not be an admin.

The resources described by this shape are special in the sense that they may only be accessed by admins, with at most one exception.

It is clear from the previous discussion that the neighborhood at least includes the following:

```
:resource2 a :resource .

:user_b :accesses :resource2 .
:user_c :accesses :resource2 .
```

This is due to the first two constraints. However, what do we do with the qualified max count? To include what is mentioned in the shape would result in adding the following triple to the neighborhood:

```
:user_c a :user .
```

Indicating that :user_c is *not* an :admin. However, this is exactly the opposite of the information we need for conformance. What we *do* need is the triple stating that :user_b is indeed an :admin. The idea here is that we want to include in our provenance all data proving that a certain node conforms to a shape. Furthermore, in a shape where only the third constraint is specified, returning nothing also seems to be a possibility for the definition of neighborhood. However, we designed our neighborhood to contain all necessary information to decide conformance, even when other parts of the graph are added. Indeed, adding the triples

```
:user_b :accesses :resource2 .
:user_c :accesses :resource2 .
```

would break this property. Having this property is technically necessary (the details are discussed in Section 5.1.4) but also useful — it allows for neighborhood engines to be flexible and add more to the provenance, while never breaking sufficiency.

This example illustrates the intricacies of defining provenance that is sufficient. Here, we discussed the (qualified) cardinality constraint components, but defining neighborhoods for all shapes requires thinking about every feature in SHACL and choosing a suitable definition for each of them while keeping in mind sufficiency and our two design principles: determinism and minimality.

Shape Fragments

This definition of neighborhoods can be viewed as an additional semantics for SHACL defining a retrieval mechanism. We call these retrieval semantics *shape fragments*.

Given a shape and a data graph, the shape fragment is simply the neighborhood of all nodes conforming to that shape. When there are targeting declarations, we only consider the nodes that are targeted (and include the information that targets them). It can similarly be defined for sets of shapes, or for shapes graphs.

One can imagine the usefulness of such a retrieval mechanism. When dealing with large RDF graphs, a shapes graph may only describe part of it that is relevant to the intended usage. Retrieving the shape fragment of the data graph then gives us a (possibly) smaller RDF graph which is easier to process, and still satisfies the constraints while containing relevant data.

1.5 Thesis outline

This thesis is structured as follows. In Chapter 2 we discuss our formalization of SHACL as a logic and the relation between that logic with real SHACL. Chapter 3 builds on this logical formalization and fills a gap in the SHACL specification by defining recursive semantics. We compare our definitions with the ones proposed in the literature. We then continue with Chapter 4 where we study the expressiveness of some of the non-standard SHACL features, and also for some light extensions of equality and disjointness tests. In Chapter 5, we propose provenance semantics for SHACL through our notion of neighborhoods. Then, we establish the sufficiency property for it. Furthermore, we provide a translation for the provenance semantics of SHACL to SPARQL and run experiments to argue the feasibility of such a system. Finally, we conclude in Chapter 6.

1.6 Publications

The contents of this thesis are based on the following publications:

- Chapter 2 is based on

  [20] Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. SHACL: A description logic in disguise. In G. Gottlob, D. Inclezan, and M. Maratea, editors, *Logic Programming and Nonmonotonic Reasoning*, pages 75–88. Springer International Publishing, 2022

- Chapter 3 is based on

  [18] Bart Bogaerts and Maxime Jakubowski. Fixpoint semantics for recursive SHACL. In A. Formisano, Y.A. Liu, et al., editors, *Proceedings 37th International Conference on Logic Programming (Technical Communications)*, volume 345 of *Electronic Proceedings in Theoretical Computer Science*, pages 41–47, 2021

- Chapter 4 is based on

  [19] Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. Expressiveness of SHACL features and extensions for full equality and disjointness tests. *Log. Methods Comput. Sci.*, 20(1), 2024. `doi:10.46298/LMCS-20(1:16)2024`

- Chapter 5 is based on

[33] Thomas Delva, Anastasia Dimou, Maxime Jakubowski, and Jan Van den Bussche. Data provenance for SHACL. In J. Stoyanovich, J. Teubner, et al., editors, *Proceedings 26th International Conference on Extending Database Technology*, pages 285–297. openproceedings.org, 2023

Here, I want to acknowledge and thank my co-authors Thomas Delva for performing many of the early experiments, and Anastasia Dimou for the early discussions and crystallizing the ideas of Shape Fragments.

The following publications and resources were also generated as part of this PhD research, but do not fall within the scope of this thesis:

[98] Maarten Vandenbrande, Maxime Jakubowski, Pieter Bonte, Bart Buelens, Femke Ongenae, and Jan Van den Bussche. POD-QUERY: schema mapping and query rewriting for solid pods. In Irini Fundulaki, Kouji Kozaki, Daniel Garijo, and José Manuél Gómez-Pérez, editors, *Proceedings of the ISWC 2023 Posters, Demos and Industry Tracks*, volume 3632 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2023

[21] Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. Postulates for provenance: Instance-based provenance for first-order logic, 2024. ACM Symposium on Principles of Database Systems, June 2024, to appear

[54] Maxime Jakubowski. SPARQL RML Rewriter, January 2024. URL: `https://github.com/MaximeJakubowski/SRR`, `doi:10.5281/zenodo.10462628`

[53] Maxime Jakubowski. SHACL Logical Syntax Parser, January 2024. URL: `https://github.com/MaximeJakubowski/sls_project`, `doi:10.5281/zenodo.10462613`

1.7 Funding acknowledgements

The work in this PhD was funded by the Flanders AI Research Program.

2

Formalization

In this chapter we discuss the formalization of SHACL, and argue that viewing SHACL as a logic is natural. We build from the formalization of shapes in SHACL proposed by Corman, Reutter and Savkovic [30]. That work has revealed striking similarities between *shapes* and *concept expressions* from Description Logics (DLs) [12]. The similarities between SHACL and DLs run even deeper when we account for *shape names* and *targeting*, which is the actual mechanism to express constraints over RDF graphs. This connection will be further discussed in Section 2.3.

Our formalization differs from existing work in a couple of small but important ways. First, we explicitly make use of a first-order interpretation, rather than a graph. Second, the semantics for SHACL we develop would be called a "natural" semantics in database theory [2]: variables always range over the universe of all possible nodes. The use of the natural semantics avoids some anomalies that crop up in the literature, where often an "active-domain" semantics is adopted instead, in which variables range only over the set of nodes actually occurring in the input graph. Unfortunately, such a semantics does not work well with constants. The problem is that a constant mentioned in a shape may or may not actually occur in the input graph. As a result, some semantics proposed in the literature violate familiar logic laws like De Morgan's law [39]. This is troublesome, since automated tools (and humans!) that generate and manipulate logic formulas may reasonably and unwittingly assume these laws to hold. Also other research papers (see Remark 2.9) contain flaws related to not taking into account nodes that *do not* occur in the graph. This highlights the importance of taking a logical perspective on SHACL.

Our design choices are grounded in real SHACL: with each of them we will provide actual SHACL specifications that prove that SHACL validators indeed behave in the way we expect. All our examples have been tested on three implementations: Apache Jena SHACL[1] (using their Java library) TopBraid SHACL[2] (using their Java library as well as their online playground), and Zazuko[3] (using their online playground).

Our SHACL examples will assume the following prefixes are defined:

```
@prefix : <https://www.mjakubowski.info/vocabulary/>
@prefix sh: <http://www.w3.org/ns/shacl#>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

[1] https://jena.apache.org/documentation/shacl/index.html

[2] https://shacl.org/playground/

[3] https://shacl-playground.zazuko.com/

2.1 The logical perspective

In this section, we begin with the formal development of the logical core of SHACL, focusing primarily on the structural constraints. Later, in Section 2.4, we will expand this formalization into a version that corresponds fully with real SHACL on the conformance semantics. We define shape expressions, shape schemas, and conformance. Our point of departure is the treatment by Andreşel et al. [6], which we adapt and extend to our purposes.

From the outset we assume three disjoint, infinite universes N, S, and P of *node names*, *shape names*, and *property names*, respectively.[4]

We define *path expressions* E and *shapes* φ by the following grammar:

$$\begin{aligned} E &::= \mathit{id} \mid p \mid E^- \mid E_1/E_2 \mid E_1 \cup E_2 \mid E^* \\ \varphi &::= \top \mid \mathit{hasShape}(s) \mid \mathit{hasValue}(c) \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \\ &\qquad \geq_n E.\varphi \mid \mathit{eq}(p,E) \mid \mathit{disj}(p,E) \mid \mathit{closed}(Q) \end{aligned}$$

where p, s, and c stand for property names, shape names, and node names, respectively, n stands for nonzero natural numbers, and Q stands for finite sets of property names. As we will formalize below, every property name evaluates to a binary relation, as does each path expression. In the path expressions, id represents the identity relation, p^- represents the inverse relation of p, E_1/E_2 represents composition of binary relations, and E^* the reflexive-transitive closure of E. We will often use the following abbreviations: $E?$ for $E \cup \mathit{id}$; $\bot$ for $\neg\top$; $\leq_n E.\varphi$ for $\neg \geq_{n+1} E.\varphi$; $\forall E.\varphi$ for $\neg \geq_1 E.\neg\varphi$; and $\exists E.\varphi$ for $\neg\forall E.\neg\varphi$.

Remark 2.1. In our formalization, a path expression can be 'id'. Later, in Chapter 4, we show in Lemma 4.3 that every path expression is equivalent to id, $E' \cup \mathit{id}$ or E', where E' does not use id. In real SHACL, it is possible to write $E' \cup \mathit{id}$ using "zero-or-one" path expressions. Explicitly writing id is not possible, but this poses no problem. Path expressions can only appear in counting quantifiers, equality and disjointness shapes. The shape $\geq_n \mathit{id}.\varphi$ is clearly equivalent to φ if $n = 1$, otherwise, it is equivalent to $\neg\top$. The shapes $\mathit{eq}(E,p)$ or $\mathit{disj}(E,p)$ where E is id are implicitly expressible in SHACL by writing the equality or disjointness constraint in node shapes, rather than property shapes.

As we will see, shapes, which represent unary predicates, will evaluate to a subset of the domain. The three last expressions are probably the least familiar. Equality ($\mathit{eq}(p,E)$) means that there are outgoing p-edges (edges labeled p) exactly to those nodes for which there is a path satisfying the expression E (defined below). Disjointness ($\mathit{disj}(p,E)$) means that there are *no* outgoing p-edges to which there is also a path satisfying E. Closedness is also a typical SHACL feature: $\mathit{closed}(Q)$ states that there are no outgoing edges about any predicates other than those in Q.

A *vocabulary* Σ is a subset of $N \cup S \cup P$. A path expression or shape is said to be *over* Σ if it only uses symbols from Σ. On the most general logical level, shapes are evaluated in *interpretations*. We recall the familiar definition: An interpretation I over Σ consists of

1. a set Δ^I, called the *domain* of I;
2. for each constant $c \in \Sigma$, an element $[\![c]\!]^I \in \Delta^I$;
3. for each shape name $s \in \Sigma$, a subset $[\![s]\!]^I$ of Δ^I; and
4. for each property name $p \in \Sigma$, a binary relation $[\![p]\!]^I$ on Δ^I.

On any interpretation I as above, every path expression E over Σ evaluates to a binary relation $[\![E]\!]^I$ on Δ^I, and every shape φ over Σ evaluates to a subset of Δ^I, as defined in

[4] In real SHACL, the disjointness assumption does not hold. However, it is only made for simplicity of notation.

Table 2.1: Semantics of a path expression E in an interpretation I over Σ.

E	$[\![E]\!]^I$
id	$\{(a,b) \in \Delta^I \times \Delta^I \mid a = b\}$
p^-	$\{(a,b) \in \Delta^I \times \Delta^I \mid (b,a) \in [\![p]\!]^I\}$
$E_1 \cup E_2$	$[\![E_1]\!]^I \cup [\![E_2]\!]^I$
E_1/E_2	$\{(a,b) \in \Delta^I \times \Delta^I \mid \exists c : (a,c) \in [\![E_1]\!]^I \wedge (c,b) \in [\![E_2]\!]^I\}$
E^*	the reflexive-transitive closure of $[\![E]\!]^I$

Table 2.2: Semantics of a shape φ in an interpretation I over Σ. For a set X, we use $\sharp X$ to denote its cardinality. For a binary relation R and an element a, we use $R(a)$ to denote the set $\{b \in \Delta^I \mid (a,b) \in R\}$.

φ	$[\![\varphi]\!]^I$
$\top$	Δ^I
$hasShape(s)$	s^I
$hasValue(c)$	$\{c^I\}$
$\varphi_1 \wedge \varphi_2$	$[\![\varphi_1]\!]^I \cap [\![\varphi_2]\!]^I$
$\varphi_1 \vee \varphi_2$	$[\![\varphi_1]\!]^I \cup [\![\varphi_2]\!]^I$
$\neg\varphi_1$	$\Delta^I \setminus [\![\varphi_1]\!]^I$
$\geq_n E.\varphi_1$	$\{a \in \Delta^I \mid \sharp([\![\varphi_1]\!]^I \cap [\![E]\!]^I(a)) \geq n\}$
$eq(p,E)$	$\{a \in \Delta^I \mid [\![p]\!]^I(a) = [\![E]\!]^I(a)\}$
$disj(p,E)$	$\{a \in \Delta^I \mid [\![p]\!]^I(a) \cap [\![E]\!]^I(a) = \emptyset\}$
$closed(Q)$	$\{a \in \Delta^I \mid [\![p]\!]^I(a) = \emptyset$ for every $p \in \Sigma \setminus Q\}$

Tables 2.1 and 2.2. In Table 2.2 we use the notation $[\![E]\!]^I(a)$ to denote the set $\{b \mid (a,b) \in [\![E]\!]^I\}$. Given an interpretation I, a node $a \in \Delta^I$ and a shape φ, we sometimes write $I, a \models \varphi$ when $a \in [\![\varphi]\!]^I$.

Example 2.2. We can write the examples from Section 1.1 in our formalization. For example, `:basicUserShape` corresponds to the expression:

$$\exists\texttt{:accesses}.\top \wedge \leq_0 \texttt{:creates}.\top$$

While `:authorizedUserShape` corresponds to the expression:

$$\exists(\texttt{:approves}^-)^*.\exists\texttt{rdf:type}.hasValue(\texttt{:Admin})$$

To see the difference in semantics, `:altAuthorizedUserShape` can be written as:

$$\exists\texttt{:approves}^-.\top \wedge \forall(\texttt{:approves}^-)^*.\exists\texttt{rdf:type}.hasValue(\texttt{:Admin})$$

We define a *shape schema* $\mathcal{S}$ over Σ as a tuple (D,T) with D a finite set of *shape definitions* $s \leftarrow \varphi$ with $s \in S$ and φ a shape expression. For now, we only consider *non-recursive* shape definitions where shapes are not defined in terms of themselves. In Chapter 3, we extend our definition to include recursive shapes. Next, T is a finite set of inclusion statements, or *target declarations* of the form $\tau \subseteq s$ with $s \in S$ and where τ is a special kind of shape expression we call a *targeting query* or *target expression.* The W3C recommendation defines five types of target declarations that correspond to a shape expression. These will be discussed in Section 2.4. For now, it suffices to know that these target declarations from the recommendation can be viewed as shape expressions. Throughout this thesis, unless explicitly mentioned, we consider *generalized shape schemas* where τ can be any shape, not just the ones allowed by the recommendation. Furthermore, we sometimes also write a shape expression on the right-hand side of a target declaration, instead of only shape names.

Example 2.3. We can now write the example schema consisting of `:authorizedUserShape` from Section 1.1 as follows. The set of definitions D consists of:

$$\texttt{:authorizedUserShape} \leftarrow \exists(\texttt{:approves}^{-})^{*}.\mathit{hasShape}(\texttt{:isAdminShape})$$
$$\texttt{:isAdminShape} \leftarrow \exists\texttt{rdf:type}.\mathit{hasValue}(\texttt{:Admin})$$

The target inclusion is:

$$\exists\texttt{:approves}.\top \subseteq \texttt{:authorizedUserShape}$$

If $\mathcal{S} = (D, T)$ is a shape schema over Σ and I an interpretation over $\Sigma \setminus S$, then there is a unique interpretation $I \diamond D$ that agrees with I outside of S and that satisfies D, i.e., such that for every expression $s \leftarrow \varphi \in D$, $[\![s]\!]^{I \diamond D} = [\![\varphi]\!]^{I \diamond D}$. We say that I *conforms to* $\mathcal{S}$, denoted by $I \models \mathcal{S}$, if $[\![\tau]\!]^{I \diamond D}$ is a subset of $[\![s]\!]^{I \diamond D}$, for every statement $\tau \subseteq s$ in T. In other words, I conforms to $\mathcal{S}$ if there exists an interpretation that satisfies $D \cup T$ that coincides with I on $N \cup P$.

Remark 2.4. There are some notable differences between real SHACL shapes graphs and our shape schemas. First, for now, we take abstraction of some features of real SHACL, such as checking data types like numbers and strings.

Second, in real SHACL not every shape name needs to occur in the left-hand side of a defining rule. The default that is taken in real SHACL is that shapes without a definition are *always satisified.* On the logical level, this means that for every shape s name that has no explicit definition, a definition $s \leftarrow \top$ is implicitly assumed. The following example that illustrates that our chosen default indeed corresponds to actual SHACL.

Example 2.5. The following SHACL shape `:trueShape` states that all nodes with an `:r`-edge must conform to the `:noDef` and `:alsoNoDef` shapes which we do not define.

```
:trueShape a sh:NodeShape ;
    sh:and ( :noDef :alsoNoDef ) .
:trueShape sh:targetSubjectsOf :r .
```

In our formal notation, this shapes graph corresponds to the shape schema

$$\texttt{:trueShape} \leftarrow \mathit{hasShape}(\texttt{:noDef}) \wedge \mathit{hasShape}(\texttt{:alsoNoDef})$$
$$\exists\texttt{:r}.\top \subseteq \texttt{:trueShape}$$

where the first line is the definition of `:trueShape`, and the second line its target.

When validating a graph containing only the triple `:a :r :b` (as we will show later, this corresponds to an interpretation in which the property name `:r` has the interpretation $\{(\texttt{:a}, \texttt{:b})\}$ and the interpretation of all other property names is empty), and thus targeting the node `:a`, it validates without violation. This supports our observation that shapes without an explicit definition are assumed to be satisfied by all nodes (i.e., are interpreted as $\top$).

To further strengthen this claim, if instead we consider the SHACL shapes graph

```
:trueShape a sh:NodeShape ;
    sh:not :noDef .
:trueShape sh:targetSubjectsOf :r .
```

i.e., the shape schema

$$\texttt{:trueShape} \leftarrow \neg\mathit{hasShape}(\texttt{:noDef})$$
$$\exists\texttt{:r}.\top \subseteq \texttt{:trueShape}$$

validation on the same graph yields the validation error that “node `:a` does not satisfy `:trueShape` since it has shape `:noDef`”.

2.2 From graphs to interpretations

Up to this point, we have discussed the logical semantics of SHACL, i.e., how to evaluate a SHACL expression in a standard first-order interpretation. However, in practice, SHACL is not evaluated on interpretations but on RDF graphs. In this section, we show precisely and unambiguously how to go from an RDF graph to a logical interpretation (in such a way that the actual SHACL semantics coincides with what we described above). A *graph* is a finite set of *facts*, where a fact is of the form $p(a, b)$, with p a property name and a and b node names. We refer to the node names appearing in a graph G simply as the *nodes* of G; the set of nodes of G is denoted by N_G. A pair (a, b) with $p(a, b) \in G$ is referred to as an *edge*, or a *p-edge*, in G. The set of p-edges in G is denoted by $[\![p]\!]^G$ (this set might be empty).

We want to be able to evaluate *any* shape on *any* graph (independently of the vocabulary the shape is over). Thereto, we will unambiguously associate, to any given graph G, an interpretation I over $N \cup P$ as follows:

- Δ^I equals N (the universe of all node names).
- $[\![c]\!]^I$ equals c itself, for every node name c.
- $[\![p]\!]^I$ equals $[\![p]\!]^G$, for every property name p.

If I is the interpretation associated to G, we use $[\![E]\!]^G$ and $[\![\varphi]\!]^G$ to mean $[\![E]\!]^I$ and $[\![\varphi]\!]^I$, respectively.

RDF also has a model-theoretic semantics [80]. These semantics reflect the view of an RDF graph as a basic ontology or logical theory, as opposed to the view of an RDF graph as an interpretation. Since the latter view is the one followed by SHACL, it is thus remarkable that SHACL effectively ignores the W3C-recommended semantics of RDF.

Remark 2.6. Andreşel et al. [6] define $[\![\varphi]\!]^G$ a bit differently. For a constant c, they define $[\![\mathit{hasValue}(c)]\!]^G = \{c\}$ like we do. For all other constructs, however, they define $[\![\varphi]\!]^G$ to be $[\![\varphi]\!]^I$, but with the domain of I taken to be N_G, rather than N. In that approach, if $c \notin N_G$, $[\![\neg\neg \mathit{hasValue}(c)]\!]^G$ would be empty rather than $\{c\}$ as one would expect. For another illustration, still assuming $c \notin N_G$, $[\![\neg(\neg\varphi \land \neg \mathit{hasValue}(c))]\!]^G$ would be $[\![\varphi]\!]^G$ rather than $[\![\varphi]\!]^G \cup \{c\}$, so De Morgan's law would fail. The next examples shows that actual SHACL implementations indeed coincide with our semantics.

Example 2.7. The following SHACL shape `:notnotShape` states that it cannot be so that the node `:myNode` is different from itself (i.e., that it must be equal to itself, but specified with a double negation).

```
:notnotShape a sh:NodeShape ;
    sh:not [ sh:not [ sh:hasValue :myNode ] ] .
:notnotShape sh:targetNode :myNode .
```

In our formal notation, this shapes graph corresponds to the shape schema

$$\texttt{:notnotShape} \leftarrow \neg\neg \mathit{hasValue}(\texttt{:myNode})$$
$$\mathit{hasValue}(\texttt{:myNode}) \subseteq \texttt{:notnotShape}$$

Clearly, this shape should validate every graph, also graphs in which the node `:myNode` is not present and it indeed does so in all SHACL implementations we tested. This supports our choice of the natural semantics, rather than the active domain semantics from the literature [6, 61]. Indeed, in that semantics, this shape will never validate any graph because the right-hand side of the inclusion will be evaluated to be the empty set.

Example 2.8. Another example, to show that the natural semantics correctly formalizes SHACL, is the one where previous semantics do not respect the De Morgan's laws:

```
:demorganShape a sh:NodeShape ;
    sh:not [
        sh:and (
            [ sh:not [
                  sh:path :r ;
                  sh:minCount 1 ] ]
            [ sh:not [ sh:hasValue :myNode ] ] ) ] .
:demorganShape sh:targetNode :myNode .
```

This shapes graph corresponds to the shape schema

$$\begin{aligned}&\texttt{:demorganShape} \leftarrow \neg(\neg\exists\texttt{:r}.\top \wedge \neg \mathit{hasShape}(\texttt{:myNode}))\\&\mathit{hasValue}(\texttt{:myNode}) \subseteq \texttt{:demorganShape}\end{aligned}$$

In the formalism of Andreşel et al. [6], this schema does not validate on graphs that do not mention the node `:myNode`, but it does in all tested SHACL implementations.

Remark 2.9. The use of active domain semantics has also introduced some errors in previous work. For instance, in the work by Leinberger et al. [61, Theorem 1] is factually incorrect. The problem originates with their notion of *faithful assignment*, which was introduced by Corman et al. This notion is defined in an active-domain fashion, only considering nodes actually appearing in the graph. For a concrete counterexample to that theorem, consider a single shape named s defined as $\exists r.\top$, with target $\mathit{hasValue}(b)$. In our terminology, this means that

$$\begin{aligned}D &= \{s \leftarrow \exists r.\top\}, \text{ and}\\T &= \{\mathit{hasValue}(b) \subseteq s\}.\end{aligned}$$

On a graph G in which b does not appear, we can assign $\{s\}$ to all nodes from G with an outgoing r-edge (meaning that all these nodes satisfy s and no other shape (names)), and assign the empty set to all other nodes (meaning that all other nodes do not satisfy any shape). According to the definition, this is a faithful assignment. However, the inclusion $\mathit{hasValue}(b) \subseteq s$ is not satisfied in the interpretation they construct from this assignment, thus violating their Theorem 1.

The anomalies in previous work will only occur in corner cases where the shape schema mentions nodes that not occur in the graph. However, simply disallowing these corner cases is not a suitable solution. Indeed, shape schemas are designed to validate graphs not known at design-time, and it should be possible to check conformance of *any* graph with respect to *any* shape schema. As the following example shows, it makes sense that a graph should conform to a schema in case a certain node does *not* occur in the graph (or does not occur in a certain context), and that the natural semantics indeed coincides with the behaviour of SHACL validators in such cases.

Example 2.10. Consider a schema $\mathcal{S}$ with $D = \emptyset$ and T consisting of a single inclusion

$$\mathit{hasValue}(\mathit{MarcoMaratea}) \subseteq \neg\exists(\mathit{author}/\mathit{venue}).\mathit{hasValue}(\mathit{LPNMR22}),$$

which states that Marco Maratea (one of the LPNMR PC chairs) does not author any LPNMR paper. If Marco Maratea does not occur in the list of of accepted papers, this list should clearly[5] conform to this schema. This example can be translated into actual SHACL, as shown in Listing 2.1.

We simply give the name `:notAnAuthorShape` to the shape that holds for all nodes that do not author any LPNMR paper and subsequently enforce that Marco Maratea satisfy this shape. We see that indeed, in accordance with our proposed semantics, graphs without a node `:MarcoMaratea` validate with respect to this SHACL specification.

Listing 2.1: The `:notAnAuthorShape` from Example 2.10 in real SHACL

```
:notAnAuthorShape a sh:NodeShape ;
  sh:not [
    sh:path (:author :venue) ;
    sh:qualifiedValueShape [ sh:hasValue :LPNMR22 ] ;
    sh:qualifiedMinCount 1 ] .
ex:notAnAuthorShape sh:targetNode :MarcoMaratea .
```

The definition of I makes — completely independent of the actual language features of SHACL — a couple of assumptions explicit:

- First of all, SHACL uses unique names assumptions (UNA): each constant is interpreted in I as a different domain element.
- Secondly, if $p(a,b)$ does not occur in the graph, it is assumed to be *false*. However, if a node c does not occur anywhere in the graph, it is not assumed to not exist: the domain of I is infinite!

Rephrasing this: SHACL makes the Closed World Assumption (CWA) on predicates (formally, $[\![p]\!]^I = [\![p]\!]^G$), but not on objects (formally, $\Delta^I = N$); the predicates are fixed by the graph, while the objects can be any node from the domain.

Effective evaluation Since the interpretation defined from a graph has the infinite domain N, it is not immediately clear that shapes can be effectively evaluated over graphs. As indicated above, however, we can reduce to a finite interpretation. Let $\Sigma \subseteq N \cup P$ be a finite vocabulary, let φ be a shape over Σ, and let G be a graph. From G we define the interpretation $I_\star$ over Σ just like I above, except that the domain of $I_\star$ is not N but rather

$$N_G \cup (\Sigma \cap N) \cup \{\star\},$$

where $\star$ is an element not in N. We use $[\![\varphi]\!]^G_\star$ to denote $[\![\varphi]\!]^{I_\star}$ and find:

Theorem 2.11 (Adapted Theorem 5.6.1 from [2])**.** *For every $x \in N_G \cup (\Sigma \cap N)$, we have $x \in [\![\varphi]\!]^G$ if and only if $x \in [\![\varphi]\!]^G_\star$. For all other node names x, we have $x \in [\![\varphi]\!]^G$ if and only if $\star \in [\![\varphi]\!]^G_\star$.*

Hence, for any shape schema $\mathcal{S}$, I conforms to $\mathcal{S}$ if and only if $I_\star$ does.

Theorem 2.11 shows that conformance can be performed by finite model checking.

2.3 SHACL, Description Logics, and OWL

The way we formalized SHACL in the previous section makes the correspondence to a description logic evident. A *shape schema* $\mathcal{S} = (D, T)$ can be seen as a *TBox* consisting of two parts. First, the shape definitions D give us an *acyclic TBox* consisting of all the formulas

$$s \equiv \varphi$$

for each statement $s \leftarrow \varphi$ in D.

Similarly, each inclusion statement $\tau \subseteq s$ from the target declarations T gives us the formulas:

$$\tau \sqsubseteq s$$

[5] Technically, the standard is slightly ambiguous with respect to nodes not occurring in the data graph.

As such, in description logic terminology, a shape schema consists of two parts: an acyclic TBox (defining the shapes in terms of the given input graph) and a general TBox (containing the actual integrity constraints).

Despite the strong similarity between SHACL and DLs, and despite the fact that in a couple of papers, SHACL has been formalized in a way that is extremely similar to description logics [6,30,61], this connection is not recognized in the community. In fact, some important stakeholders in SHACL even wrote the following in a blog post explaining why they use SHACL, rather than OWL:

> OWL was inspired by and designed to exploit 20+ years of research in Description Logics (DL). This is a field of mathematics that made a lot of scientific progress right before creation of OWL. I have no intention of belittling accomplishments of researchers in this field. However, there is little connection between this research and the practical data modeling needs of the common real world software systems. [75]

thereby suggesting that SHACL and DLs are two completely separated worlds. On top of that, SHACL is presented by some stakeholders [92] as an alternative to the Web ontology language OWL [67], which is based on the description logic $\mathcal{SROIQ}$ [52].

This naturally begs the question: which misunderstanding is it that drives this wedge between communities? How can we explain this discrepancy from a mathematical perspective (thereby patently ignoring strategic, economic, social, and other aspects that play a role).

Our answer is that there are two important differences between OWL and SHACL that deserve attention. These differences, however, do not contradict our central claim here, which is that *SHACL is a description logic.*

1. The first difference is that **in SHACL, the data graph (implicitly) represents a first-order interpretation, while in OWL, it represents a first-order theory (an ABox)**. Of course, viewing the same syntactic structure (an RDF graph) as an interpretation is very different from viewing it as a theory. While this is a discrepancy between OWL and SHACL, theories as well as interpretations exist in the world of description logic and as such, this view is perfectly compatible with our central argument. There is, however, one caveat with this claim that deserves some attention, and that is highlighted by the use of the world "implicitly". Namely, to the best of our knowledge, it is never mentioned that the data graph simply represents a standard first-order interpretation, and it has not been made formal what *exactly* the interpretation is that is associated to a graph. We made this correspondence between graphs an interpretations explicit in Section 2.2. However, in real SHACL, instead, the language features are typically evaluated *directly* on the data graph. There are several reasons why we believe it is important to make this translation of a graph into an interpretation *explicit.*

    - This translation makes *the assumptions SHACL makes about the data* explicit. For instance, it is often informally stated that "SHACL uses closed-world assumptions" [55]; we will make this statement more precise: SHACL uses closed-world assumptions with respect to the relations, but open-world assumptions on the domain.
    - Once the graph is eliminated, we are in familiar territory. In the field of description logics a plethora of language features have been studied. It now becomes clear how to add them to SHACL, if desired. The 20+ years of research mentioned in [75] suddenly become directly applicable to SHACL.

2. The second difference, which closely relates to the first, is that **OWL and SHACL have a different (default) inference task**: the standard inference task at hand

> in OWL is *deduction*, while in SHACL, the main task is checking the conformance of RDF graphs against shape schemas. In logical terminology, this is evaluating whether a given interpretation satisfies a theory (TBox), i.e., this is the task of *model checking.*
>
> Of course, the fact that a different inference task is typically associated with these languages does not mean that their logical foundations are substantially different. Furthermore, other researchers [61,69,70] have investigated tasks such as *satisfiability* and *containment* (which are among the tasks typically studied in DLs) for SHACL, making it all the more obvious that the field of description logics has something to offer for studying properties of SHACL.

This connection with description logics can even inform us on the behaviour of SHACL. While Theorem 2.11 shows that conformance can be performed by finite model checking, other tasks typically studied in DLs are not decidable; this can be shown with a small modification of the proof of undecidability of the description logic $\mathcal{ALRC}$, as detailed by Schmidt-Schauß [84].

Theorem 2.12. *Consistency of a shape schema (i.e., the question whether or not some I conforms to $\mathcal{S}$) is undecidable.*

Following description logic traditions, decidable fragments of SHACL have been studied already; for instance Leinberger et al. [61] disallow equality, disjointness, and closedness in shapes, as well as union and Kleene star in path expressions.

2.4 Correspondence with the recommendation

When we talk about SHACL and use a formalization, like our logic, it is important that results about the logic translate back to the tool we are actually interested in. In some sense, formalizing SHACL purely as a logic gives us the freedom to study it for its own sake — the logic onto itself is interesting: a description logic with some unfamiliar features like equality, disjointness and closedness. But in the other direction, it would be ideal to study this logic and get results that are of interest to SHACL users. That is what this section is about: pinpointing exactly how formal SHACL relates to real SHACL.

First, we start with undoing some of the simplifications we made in the previous sections. We will go over the logic again, but now in such a way that it includes all details that are relevant to practical conformance checking.

We partition the set of nodes N into three pairwise disjoint infinite sets I, L, and B of *IRIs*, *literals*, and *blank nodes*, respectively. Literals may have a "language tag" [79]. We abstract this by assuming an antireflexive relation $\sim$ on L, where $l \sim l'$ represents that l and l' are distinct literals with the same language tag. Moreover, we need to introduce a strict partial order $<$ on N, where the focus is on comparing numeric values, strings, dateTime values etc. This partial order is based on the one from the SPARQL recommendation for comparing RDF terms. More information can be found there [50].

Now, we will define an RDF graph in more detail. The correspondence with interpretations is still straightforward. An *RDF triple* (s, p, o) is an element of $(I \cup B) \times I \times N$. We refer to the elements of the triple as the subject s, the property p, and the object o. An *RDF graph* G is a finite set of RDF triples.

SHACL can do many tests on individual nodes, such as testing whether a node is a literal, or testing whether an IRI matches some regular expression. We abstract this by assuming a set Ω of *node tests*; for any node test t and node a, we assume it is well-defined whether or not *a satisfies t*.

The grammar of the shapes is slightly extended in features, but restricted in structure. Most notably, path expressions don't have an explicit *id* feature. It is hidden in the E?

Table 2.3: Conditions for conformance of a node to a shape.

φ	$\mathcal{S}, G, a \models \varphi$ if:
$\mathit{hasValue}(c)$	$a = c$
$\mathit{test}(t)$	a satisfies t
$\mathit{hasShape}(s)$	$\mathcal{S}, G, a \models \mathit{def}(s, \mathcal{S})$
$\geq_n E.\psi$	$\sharp\{b \in [\![E]\!]^G(a) \mid \mathcal{S}, G, b \models \psi\} \geq n$
$\leq_n E.\psi$	$\sharp\{b \in [\![E]\!]^G(a) \mid \mathcal{S}, G, b \models \psi\} \leq n$
$\forall E.\psi$	every $b \in [\![E]\!]^G(a)$ satisfies $\mathcal{S}, G, b \models \psi$
$\mathit{eq}(F, p)$	the sets $[\![F]\!]^G(a)$ and $[\![p]\!]^G(a)$ are equal
$\mathit{disj}(F, p)$	the sets $[\![F]\!]^G(a)$ and $[\![p]\!]^G(a)$ are disjoint
$\mathit{closed}(P)$	for all triples $(a, p, b) \in G$ we have $p \in P$
$\mathit{lessThan}(E, p)$	$b < c$ for all $b \in [\![E]\!]^G(a)$ and $c \in [\![p]\!]^G(a)$
$\mathit{lessThanEq}(E, p)$	$b \leq c$ for all $b \in [\![E]\!]^G(a)$ and $c \in [\![p]\!]^G(a)$
$\mathit{uniqueLang}(E)$	$b \not\sim c$ for all $b \neq c \in [\![E]\!]^G(a)$.

construct, which is defined as $E \cup \mathit{id}$. Furthermore, id can only explicitly be used in the equality and disjointness tests. The grammar is as follows:

$$\begin{aligned} E &::= p \mid E^- \mid E_1/E_2 \mid E_1 \cup E_2 \mid E? \mid E^* \\ F &::= \mathit{id} \mid E \\ \varphi &::= \top \mid \mathit{hasShape}(s) \mid \mathit{hasValue}(c) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \\ &\quad\mid \geq_n E.\varphi \mid \leq_n E.\varphi \mid \forall E.\varphi \mid \mathit{eq}(F, p) \mid \mathit{disj}(F, p) \mid \mathit{closed}(Q) \\ &\quad\mid \mathit{lessThan}(E, p) \mid \mathit{lessThanEq}(E, p) \mid \mathit{uniqueLang}(E) \mid \mathit{test}(t) \end{aligned}$$

with $s \in I \cup B$; $t \in \Omega$; $c \in N$; n a natural number; E a path expression; $p \in I$; and $Q \subseteq I$ finite.

The features common between the grammar here, and the grammar from Section 2.1 have the same semantics as before. However, here, we do not work with first-order interpretations but directly on RDF graphs. The added features make use of the details we added in this section. In order to define the semantics of shapes and shape schemas, we first recall that a path expression E evaluates on an RDF graph G to a binary relation on N, denoted by $[\![E]\!]^G$ and defined as follows:

- $[\![p]\!]^G = \{(a, b) \mid (a, p, b) \in G\}$;
- $[\![E^-]\!]^G = \{(b, a) \mid (a, b) \in [\![E]\!]^G\}$;
- $[\![E?]\!]^G = \{(a, a) \mid a \in N\} \cup [\![E]\!]^G$;
- $[\![E_1 \cup E_2]\!]^G = [\![E_1]\!]^G \cup [\![E_2]\!]^G$;
- $[\![E_1/E_2]\!]^G = \{(a, c) \mid \exists b : (a, b) \in [\![E_1]\!]^G \;\&\; (b, c) \in [\![E_2]\!]^G\}$; and
- $[\![E^*]\!]^G =$ the reflexive-transitive closure of $[\![E]\!]^G$.

Finally, we also define $[\![\mathit{id}]\!]^G$, for any G, to be simply the identity relation on N.

We define the semantics for when a node v satisfies a shape φ in graph G in context of a schema $\mathcal{S}$, denoted by $\mathcal{S}, G, v \models \varphi$ in Table 2.3. We use the notation $\mathit{def}(s, \mathcal{S})$ to denote the shape expression φ associated with the shape name s from the set of shape definitions.

Remark 2.13. Curiously, SHACL provides shapes *lessThan* and *lessThanEq* but not their variants *moreThan* and *moreThanEq* (with the obvious meaning). Note that $\mathit{moreThan}(E, p)$ is not equivalent to $\neg\mathit{lessThanEq}(E, p)$.

Furthermore, we have to restrict the allowed targeting queries to the ones actually allowed by real SHACL. Given a parameter $t \in N$, these target declarations are:

Node targets which target a specific node in the graph. It corresponds to the shape expression $\mathit{hasValue}(t)$.

Class-based targets which targets all nodes in the graph of RDF-type t. It corresponds to the shape expression $\exists \text{rdf:type/rdfs:subClassOf}^{*}.\mathit{hasValue}(t)$.

Implicit class targets which targets all nodes in the graph that are of type s where s is the name of the shape associated with the target declaration. This is expressible as a class-based target declaration.

Subjects-of targets which targets all nodes in the graph that are the subject of triples with predicate t. It corresponds to the shape expression $\exists t.\top$.

Objects-of targets which targets all nodes in the graph that are the object of triples with predicate t. It corresponds to the shape expression $\exists t^{-}.\top$.

We are now ready to state the main result of this section:

Theorem 2.14. *Every formal SHACL schema can be written as a SHACL shapes graph, and vise versa.*

The proof of this Theorem will be given by two translations described in the following sections. First, in Section 2.4.1 we give a translation from real SHACL to formal SHACL. Then, in Section 2.4.2, we give a translation from formal SHACL to real SHACL.

2.4.1 Translating real SHACL to formal SHACL

In this section we define the function t which maps a SHACL shapes graph $\mathcal{G}$ to a schema (D, T). It is inspired by a similar treatment by Corman et al. [30] who translate SHACL into their formalization. Here, we try to amend some inaccuracies and be more complete towards real SHACL.

We assume that:

1. The shapes graph is well-formed[6]; and
2. All shapes are explicitly declared to be a `sh:NodeShape` or `sh:PropertyShape`.

The first assumption simply states that the RDF graph actually represents a shapes graph. The second one is for simplicity of the translation. In real SHACL, shapes do not need to be explicitly declared a node or property shape. However, it can be inferred.[7]

Let the sets $\mathcal{G}_n$, $\mathcal{G}_p$ and $\mathcal{G}_t$ respectively be the sets of node shape names, property shape names and the names of shapes that have a target declaration associated with it. Let d_x denote the set of RDF triples in $\mathcal{G}$ with x as the subject. The set of shape definitions D has a definition for every $s \in \mathcal{G}_n : s \leftarrow t_{nodeshape}(d_s)$ and a definition for every $s \in \mathcal{G}_p : s \leftarrow t_{propertyshape}(d_s)$. The set of target declarations has an inclusion statement for every $s \in \mathcal{G}_t : t_{target}(d_s) \subseteq s$. The three important functions $t_{nodeshape}(d_x)$, $t_{propertyshape}(d_x)$, and $t_{target}(d_x)$ are defined in the paragraphs below.

Remark 2.15. We treat node shapes and property shapes separately. In particular, cardinality and unique language constraints are only treated below under property shapes. Strictly speaking, however, these constraints may also be used in node shapes, where they are redundant, as the count equals one in this case. For simplicity, we assume the shapes graph does not contain such redundancies.

[6] https://www.w3.org/TR/shacl/#dfn-well-formed

[7] See paragraph 2.2 and 2.3 from the specification [86]

Defining $t_{nodeshape}(d_x)$

This function translates SHACL node shapes to shapes in the formalization. We define $t_{nodeshape}(d_x)$ to be the following conjunction:

$$t_{shape}(d_x) \wedge t_{logic}(d_x) \wedge t_{tests}(d_x) \wedge t_{value}(d_x) \wedge t_{in}(d_x) \wedge t_{closed}(d_x) \wedge t_{pair}(id, d_x) \wedge t_{languagein}(d_x)$$

where we define $t_{shape}(d_x)$, $t_{logic}(d_x)$, $t_{tests}(d_x)$, $t_{value}(d_x)$, $t_{in}(d_x)$, $t_{closed}(d_x)$, $t_{langin}(d_x)$ and $t_{pair}(id, d_x)$ in the following paragraphs.

Defining $t_{shape}(d_x)$ This function translates the Shape-based Constraint Components[8] from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:node` and `sh:property`.

We define $t_{shape}(d_x)$ to be the conjunction:

$$\bigwedge_{(x,\texttt{sh:node},y)\in d_x} \mathit{hasShape}(y) \wedge \bigwedge_{(x,\texttt{sh:property},y)\in d_x} \mathit{hasShape}(y)$$

Defining $t_{logic}(d_x)$ This function translates the Logical Constraint Components[9] from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:and`, `sh:or`, `sh:not`, `sh:xone`.

We define $t_{logic}(d_x)$ as follows:

$$\begin{gathered}\bigwedge_{(x,\texttt{sh:not},y)\in d_x} (\neg \mathit{hasShape}(y)) \wedge \\ \bigwedge_{(x,\texttt{sh:and},y)\in d_x} (\bigwedge_{z\in y} \mathit{hasShape}(z)) \wedge \\ \bigwedge_{(x,\texttt{sh:or},y)\in d_x} (\bigvee_{z\in y} \mathit{hasShape}(z)) \wedge \\ \bigwedge_{(x,\texttt{sh:xone},y)\in d_x} (\bigvee_{a\in y} (a \wedge \bigwedge_{b\in y-\{a\}} \neg \mathit{hasShape}(b)))\end{gathered}$$

where we note that the object y of the triples with the predicate `sh:and`, `sh:or`, or `sh:xone` is a SHACL list.

Defining $t_{tests}(d_x)$ This function translates the Value Type Constraint Components[10], Value Range Constraint Components[11], and String-based Constraint Components[12], with exception to the `sh:languageIn` keyword which is handled separately with the function $t_{languagein}$, from d_x to shapes from the formalization. This function covers the SHACL keywords:

> `sh:class`, `sh:datatype`, `sh:nodeKind`, `sh:minExclusive`, `sh:maxExclusive`, `sh:minLength`, `sh:maxLength`, `sh:pattern`.

We define $t_{tests}(d_x)$ as follows:

$$t_{tests'}(d_x) \wedge \bigwedge_{(x,\texttt{sh:class},y)\in d_x} \exists \texttt{rdf:type/rdfs:subclassOf}^*.\mathit{hasValue}(y)$$

[8] https://www.w3.org/TR/shacl/#core-components-shape

[9] https://www.w3.org/TR/shacl/#core-components-logical

[10] https://www.w3.org/TR/shacl/#core-components-value-type

[11] https://www.w3.org/TR/shacl/#core-components-range

[12] https://www.w3.org/TR/shacl/#core-components-string

where $t_{tests'}(d_x)$ is defined next. Let Γ denote the set of keywords just mentioned above, except for `sh:class`.

$$t_{tests'}(d_x) = \bigwedge_{c \in \Gamma} \bigwedge_{(x,c,y) \in d_x} test(\omega_{c,y})$$

where $\omega_{c,y}$ is the node test in Ω corresponding to the SHACL constraint component corresponding to c with parameter y. For simplicity, we omit the `sh:flags` for `sh:pattern`.

Defining $t_{pair}(id, d_x)$ This function translates the Property Pair Constraint Components[13] when applied to a node shape from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:equals`, `sh:disjoint`, `sh:lessThan`. `sh:lessThanOrEquals`.

We define the function $t_{pair}(id, d_x)$ as follows:

- If $\exists p : (x, \texttt{sh:lessThan}, p) \in d_x$ or $(x, \texttt{sh:lessThanEq}, p) \in d_x$, then

$$t_{pair}(id, d_x) = \bot$$

- Otherwise,

$$t_{pair}(id, d_x) = \bigwedge_{(x,\texttt{sh:equals},p) \in d_x} eq(id, p) \wedge \bigwedge_{(x,\texttt{sh:disjoint},p) \in d_x} disj(id, p)$$

Defining $t_{languagein}(d_x)$ This function translates the constraint component Language In Constraint Component[14] from d_x to shapes from the formalization. This function covers the SHACL keyword: `sh:languageIn`.

The function $t_{languagein}(E, d_x)$ is defined as follows:

$$t_{languagein}(E, d_x) = \bigwedge_{(x,\texttt{sh:languageIn},y) \in d_x} \forall E. \bigvee_{lang \in y} test(\omega_{lang})$$

where y is a SHACL list and ω_{lang} is the element from Ω that corresponds to the test that checks if the node is annotated with the language tag *lang*.

Defining other constraint components These functions translate the Other Constraint Components[15] from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:closed`, `sh:ignoredProperties`, `sh:hasValue`, `sh:in`.

We define the following functions:

$$t_{value}(d_x) = \bigwedge_{(x,\texttt{sh:hasValue},y) \in d_x} hasValue(y)$$

$$t_{in}(d_x) = \bigwedge_{(x,\texttt{sh:in},y) \in d_x} (\bigvee_{a \in y} hasValue(a))$$

Let P be the set of all properties $p \in I$ such that $(y, \texttt{sh:path}, p) \in \mathcal{G}$ where y is a property shape such that $(x, \texttt{sh:property}, y) \in d_x$ union the set given by the SHACL list specified by the `sh:ignoredProperties` parameter. Then, we define the function $t_{closed}(d_x)$ as follows:

$$t_{closed}(d_x) = \begin{cases} \top & \text{if } (x, \texttt{sh:closed}, \mathit{true}) \notin d_x \\ closed(P) & \text{otherwise} \end{cases}$$

[13] https://www.w3.org/TR/shacl/#core-components-property-pairs

[14] https://www.w3.org/TR/shacl/#LanguageInConstraintComponent

[15] https://www.w3.org/TR/shacl/#core-components-others

Defining $t_{path}(pp)$

In preparation of the next Subsection, this function translates the Property Paths to path expressions. This part of the translation deals with the SHACL keywords:

`sh:inversePath`, `sh:alternativePath`, `sh:zeroOrMorePath`,
`sh:oneOrMorePath`, `sh:zeroOrOnePath`, `sh:alternativePath`.

For an IRI or blank node pp representing a property path, we define $t_{path}(pp)$ as follows:

$$t_{path}(pp) = \begin{cases} pp & \text{if } pp \text{ is an IRI} \\ t_{path}(y)^- & \text{if } \exists y : (pp, \texttt{sh:inversePath}, y) \in \mathcal{G} \\ t_{path}(y)^* & \text{if } \exists y : (pp, \texttt{sh:zeroOrMorePath}, y) \in \mathcal{G} \\ t_{path}(y)/t_{path}(y)^* & \text{if } \exists y : (pp, \texttt{sh:oneOrMorePath}, y) \in \mathcal{G} \\ t_{path}(y)? & \text{if } \exists y : (pp, \texttt{sh:zeroOrOnePath}, y) \in \mathcal{G} \\ \bigcup_{a \in y} t_{path}(a) & \begin{array}{l}\text{if } \exists y :(pp, \texttt{sh:alternativePath}, y) \in \mathcal{G} \text{ and} \\ \qquad y \text{ is a SHACL list}\end{array} \\ t_{path}(a_1)/\ldots/t_{path}(a_n) & \text{if } pp \text{ represents the SHACL list } [a_1, \ldots, a_n] \end{cases}$$

Defining $t_{propertyshape}(d_x)$

This function translates SHACL property shapes to shapes in the formalization. Let pp be the property path associated with d_x. Let E be $t_{path}(pp)$. We define $t_{propertyshape}(d_x)$ as the following conjunction:

$$t_{card}(E, d_x) \wedge t_{pair}(E, d_x) \wedge t_{qual}(E, d_x) \wedge t_{all}(E, d_x) \wedge t_{uniquelang}(E, d_x)$$

where we define t_{card}, t_{pair}, t_{qual}, t_{all}, and $t_{uniquelang}$ in the following subsections.

Defining $t_{card}(E, d_x)$ This function translates the Cardinality Constraint Components[16]. from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:minCount`, `sh:maxCount`.

We define the function $t_{card}(E, d_x)$ as follows:

$$\bigwedge_{(x,\texttt{sh:minCount},n) \in d_x} \geq_n E.\top \wedge \bigwedge_{(x,\texttt{sh:maxCount},n) \in d_x} \leq_n E.\top$$

Defining $t_{pair}(E, d_x)$ This function translates the Property Pair Constraint Components[17] when applied to a property shape from d_x to shapes from the formalization. This function covers the following SHACL keywords: `sh:equals`, `sh:disjoint`, `sh:lessThan`, and `sh:lessThanOrEquals`.

We define the function $t_{pair}(E, d_x)$ as follows:

$$\bigwedge_{(x,\texttt{sh:equals},p) \in d_x} eq(E, p) \wedge$$
$$\bigwedge_{(x,\texttt{sh:disjoint},p) \in d_x} disj(E, p) \wedge$$
$$\bigwedge_{(x,\texttt{sh:lessThan},p) \in d_x} lessThan(E, p) \wedge$$
$$\bigwedge_{(x,\texttt{sh:lessThanOrEquals},p) \in d_x} lessThanEq(E, p)$$

[16] https://www.w3.org/TR/shacl/#core-components-count

[17] https://www.w3.org/TR/shacl/#core-components-property-pairs

Defining $t_{qual}(E, d_x)$ This function translates the Qualified Shape-based Constraint Components[18] from d_x to shapes from the formalization. This function covers the SHACL keywords:

> `sh:qualifiedValueShape`, `sh:qualifiedMinCount`, `sh:qualifiedMaxCount`, `sh:qualifiedValueShapesDisjoint`.

There are two cases: either `sh:qualifiedValueShapesDisjoint` is set to *true* or it is set to *false*.

$$t_{qual}(E, d_x) = \begin{cases} t_{sibl}(E, d_x) & \text{if } (x, \texttt{sh:qualifiedValueShapesDisjoint}, \mathit{true}) \in d_x \\ t_{nosibl}(E, d_x) & \text{otherwise} \end{cases}$$

where we define $t_{sibl}(E, d_x)$ and $t_{nosibl}(E, d_x)$ next. Let $ps = \{v \mid (v, \texttt{sh:property}, x) \in \mathcal{G}\}$. We define the set of sibling shapes

$$sibl = \{w \mid \exists v \in ps\, \exists y (v, \texttt{sh:property}, y) \in \mathcal{G} : (y, \texttt{sh:qualifiedValueShape}, w) \in \mathcal{G}\}.$$

We also define:

$$\begin{aligned} \mathcal{Q} &= \{y \mid (x, \texttt{sh:qualifiedValueShape}, y) \in d_x\} \\ \mathcal{Q}min &= \{z \mid (x, \texttt{sh:qualifiedMinCount}, z) \in d_x\} \\ \mathcal{Q}max &= \{z \mid (x, \texttt{sh:qualifiedMaxCount}, z) \in d_x\} \end{aligned}$$

We now define

$$\begin{aligned} t_{sibl}(E, d_x) = \bigwedge_{y \in \mathcal{Q}} \bigwedge_{z \in \mathcal{Q}min} \geq_z E.(\mathit{hasShape}(y) \wedge \bigwedge_{s \in sibl} \neg \mathit{hasShape}(s)) \\ \wedge \bigwedge_{y \in \mathcal{Q}} \bigwedge_{z \in \mathcal{Q}max} \leq_z E.(\mathit{hasShape}(y) \wedge \bigwedge_{s \in sibl} \neg \mathit{hasShape}(s)) \end{aligned}$$

and

$$t_{nosibl}(E, d_x) = \bigwedge_{y \in \mathcal{Q}} \bigwedge_{z \in \mathcal{Q}min} \geq_z E.\mathit{hasShape}(y) \wedge \bigwedge_{y \in \mathcal{Q}} \bigwedge_{z \in \mathcal{Q}max} \leq_z E.\mathit{hasShape}(y).$$

Defining $t_{all}(E, d_x)$ This function translates the constraint components that are not specific to property shapes, but which are applied on property shapes.

We define the function $t_{all}(E, d_x)$ to be:

$$\forall E.(t_{shape}(d_x) \wedge t_{logic}(d_x) \wedge t_{tests}(d_x) \wedge t_{in}(d_x) \wedge t_{closed}(d_x) \wedge t_{languagein}(d_x)) \wedge t_{allvalue}(E, d_x)$$

where

$$t_{allvalue}(E, d_x) = \begin{cases} \top & \text{if } \nexists v : (x, \texttt{sh:hasValue}, v) \in d_x \\ \geq_1 E.t_{value}(d_x) & \text{otherwise} \end{cases}$$

and t_{shape}, t_{logic}, t_{tests}, t_{value}, $t_{languagein}$, and t_{closed} are as defined earlier. Note the treatment of the `sh:hasValue` parameter when used in a property shape. Unlike the other definitions, it is not universally quantified over the value nodes given by E.

[18] https://www.w3.org/TR/shacl/#QualifiedValueShapeConstraintComponent

Defining $t_{uniquelang}(E, d_x)$ This function translates the constraint component Unique Lang Constraint Component[19] from d_x to shapes from the formalization. This function covers the SHACL keyword: `sh:uniqueLang`.

The function $t_{uniquelang}(E, d_x)$ is defined as follows:

$$t_{uniquelang}(E, d_x) = \begin{cases} uniqueLang(E) & \text{if } (x, \texttt{sh:uniqueLang}, true) \in d_x \\ \top & \text{otherwise} \end{cases}$$

Defining $t_{target}(d_x)$

This function translates the Target declarations to shapes from the formalization. This function covers the SHACL keywords:

`sh:targetNode`, `sh:targetClass`, `sh:targetSubjectsOf`, `sh:targetObjectsOf`.

We define the function as follows:

$$\begin{aligned} t_{target}(d_x) = & \bigvee_{(x,\texttt{sh:targetNode},y)\in d_x} hasValue(y) \vee \\ & \bigvee_{(x,\texttt{sh:targetClass},y)\in d_x} \exists \texttt{rdf:type/rdf:subclassOf}^*.hasValue(y) \vee \\ & \bigvee_{(x,\texttt{sh:targetSubjectsOf},y)\in d_x} \exists y.\top \vee \\ & \bigvee_{(x,\texttt{sh:targetObjectsOf},y)\in d_x} \exists y^-.\top \end{aligned}$$

If none of these triples are in d_x we define $t_{target}(d_x) = \bot$

2.4.2 Translating formal SHACL to real SHACL

Every shape schema in formal SHACL can be viewed as a shapes graph if the target declarations correspond to the ones supported by real SHACL. We will assume this is the case for this section.

Remark 2.16. This requirement on the target declarations is not as strict as it seems. In Chapter 4, Theorem 4.20, we show that we can translate any SHACL schema that does not use closure constraints (or recursion) to a schema that uses only target declarations present in real SHACL.

The translation will be given in two parts. First, given a shape definition $s \leftarrow \varphi$ we define the function $g_d(s, \varphi)$ that gives us the corresponding (partial) shapes graph defining shape s. Then, given a targeting declaration $\tau \subseteq s$ we define the function $g_t(s, \tau)$ which returns the triple representing the real SHACL target declaration corresponding to the inclusion statement $\tau \subseteq s$.

The general translation is as follows: given a shape schema (D, T), the shapes graph is given by:

$$\bigcup_{s\leftarrow\varphi\in D} g_d(s, \varphi) \cup \bigcup_{\tau\subseteq s\in T} g_t(s, \tau)$$

[19] https://www.w3.org/TR/shacl/#UniqueLangConstraintComponent

Defining $g_t(s, \tau)$

This function assumes τ is one of the allowed target expressions discussed at the beginning of this section. For every one of these cases, $g_t(\tau, s)$ gives us a singleton graph containing one triple expressing the target expression. We will go over the possible cases for τ:

- $g_t(s, \mathit{hasValue}(t)) = \{(s, \texttt{sh:targetNode}, t)\}$
- $g_t(s, \exists\texttt{rdf:type/rdfs:subClassOf}.\mathit{hasValue}(t)) = \{(s, \texttt{sh:targetClass}, t)\}$
- $g_t(s, \exists t.\top) = \{(s, \texttt{sh:targetSubjectsOf}, t)\}$
- $g_t(s, \exists t^-.\top) = \{(s, \texttt{sh:targetObjectsOf}, t)\}$

Defining $g_p(b, E)$

Before we start defining $g_d(s, \varphi)$ we have to define a function $g_p(b, E)$ that translates a path expression E to a RDF graph that represents the path expression according to the SHACL specification and where $b \in B$ is the blank node that represents that path expression.

Note that according to the SHACL specification, path expressions are blank nodes, except when it is simply a property. For simplification of notation, we assume it will always be a blank node (and we will translate $E = p$ to the immediate equivalent: $p \cup p$). We define the function for every possible E:

- $g_p(b, p) = g_p(b, p \cup p)$
- $g_p(b, E_1^-) = \left\{\ b\ \texttt{sh:inversePath}\ b_1\ \ .\ \right\} \cup g_p(b_1, E_1)$
- $g_p(b, E_1/E_2) = \left\{ \begin{array}{l} b\ \texttt{rdf:first}\ b_1\ \texttt{;} \\ \quad \texttt{rdf:next [rdf:first}\ b_2\ \texttt{;} \\ \qquad\qquad\qquad \texttt{rdf:rest rdf:nil] .} \end{array} \right\} \cup g_p(b_1, E_1) \cup g_p(b_2, E_2)$
- $g_p(b, E_1 \cup E_2) = \left\{\ b\ \texttt{sh:alternativePath (}b_1\ \ b_2\texttt{) .}\ \right\} \cup g_p(b_1, E_1) \cup g_p(b_2, E_2)$
- $g_p(b, E_1?) = \left\{\ b\ \texttt{sh:zeroOrOnePath}\ b_1\ \ .\ \right\} \cup g_p(b_1, E_1)$
- $g_p(b, E_1^*) = \left\{\ b\ \texttt{sh:zeroOrMorePath}\ b_1\ \ .\ \right\} \cup g_p(b_1, E_1)$

Defining $g_d(s, \varphi)$

We will give a description of the output of $g_d(s, \varphi)$ for every possible value of φ.

- $g_d(s, \top) = \left\{ \begin{array}{l} s\ \texttt{a sh:NodeShape ;} \\ \quad \texttt{sh:node [] .} \end{array} \right\}$
- $g_d(s, \mathit{hasShape}(s')) = \left\{ \begin{array}{l} s\ \texttt{a sh:NodeShape ;} \\ \quad \texttt{sh:node}\ s'\ \texttt{.} \end{array} \right\}$
- $g_d(s, \mathit{hasValue}(c)) = \left\{ \begin{array}{l} s\ \texttt{a sh:NodeShape ;} \\ \quad \texttt{sh:in (}c\texttt{) .} \end{array} \right\}$
- $g_d(s, \varphi_1 \wedge \varphi_2) = \left\{ \begin{array}{l} s\ \texttt{a sh:NodeShape ;} \\ \quad \texttt{sh:and (}b_1\ \ b_2\texttt{) .} \end{array} \right\} \cup g_d(b_1, \varphi_1) \cup g_d(b_2, \varphi_2)$
- $g_d(s, \varphi_1 \vee \varphi_2) = \left\{ \begin{array}{l} s\ \texttt{a sh:NodeShape ;} \\ \quad \texttt{sh:or (}b_1\ \ b_2\texttt{) .} \end{array} \right\} \cup g_d(b_1, \varphi_1) \cup g_d(b_2, \varphi_2)$

- $g_d(s, \neg\varphi_1) = \left\{ \begin{array}{l} s\ \texttt{a sh:NodeShape ;} \\ \quad \texttt{sh:not}\ b_1\ \ \texttt{.} \end{array} \right\} \cup g_d(b_1, \varphi_1)$
- $g_d(s, \geq_n E.\varphi_1) = \left\{ \begin{array}{l} s\ \texttt{a sh:PropertyShape ;} \\ \quad \texttt{sh:path}\ b_E\ \ \texttt{;} \\ \quad \texttt{sh:qualifiedMinCount}\ n\ \ \texttt{;} \\ \quad \texttt{sh:qualifiedValueShape}\ b_1\ \ \texttt{.} \end{array} \right\} \cup g_p(b_E, E) \cup g_d(b_1, \varphi_1)$
- $g_d(s, \leq_n E.\varphi_1) = \left\{ \begin{array}{l} s\ \texttt{a sh:PropertyShape ;} \\ \quad \texttt{sh:path}\ b_E\ \ \texttt{;} \\ \quad \texttt{sh:qualifiedMaxCount}\ n\ \ \texttt{;} \\ \quad \texttt{sh:qualifiedValueShape}\ b_1\ \ \texttt{.} \end{array} \right\} \cup g_p(b_E, E) \cup g_d(b_1, \varphi_1)$
- $g_d(s, \forall E \varphi_1) = \left\{ \begin{array}{l} s\ \texttt{a sh:PropertyShape ;} \\ \quad \texttt{sh:path}\ b_E\ \ \texttt{;} \\ \quad \texttt{sh:node}\ b_1\ \ \texttt{.} \end{array} \right\} \cup g_p(b_E, E) \cup g_d(b_1, \varphi_1)$
- $g_d(s, \mathit{eq}(p, \mathit{id})) = \left\{ \begin{array}{l} s\ \texttt{a sh:NodeShape ;} \\ \quad \texttt{sh:equals}\ p\ \ \texttt{.} \end{array} \right\}$
- $g_d(s, \mathit{eq}(p, E)) = \left\{ \begin{array}{l} s\ \texttt{a sh:PropertyShape ;} \\ \quad \texttt{sh:path}\ b_E\ \ \texttt{;} \\ \quad \texttt{sh:equals}\ p\ \ \texttt{.} \end{array} \right\} \cup g_p(b_E, E)$
- $g_d(s, \mathit{disj}(p, \mathit{id})) = \left\{ \begin{array}{l} s\ \texttt{a sh:NodeShape ;} \\ \quad \texttt{sh:disjoint}\ p\ \ \texttt{.} \end{array} \right\}$
- $g_d(s, \mathit{disj}(p, E)) = \left\{ \begin{array}{l} s\ \texttt{a sh:PropertyShape ;} \\ \quad \texttt{sh:path}\ b_E\ \ \texttt{;} \\ \quad \texttt{sh:disjoint}\ p\ \ \texttt{.} \end{array} \right\} \cup g_p(b_E, E)$
- $g_d(s, \mathit{closed}(Q)) = \left\{ \begin{array}{l} s\ \texttt{a sh:NodeShape ;} \\ \quad \texttt{sh:closed true ;} \\ \quad \texttt{sh:ignoredProperties}\ Q\ \ \texttt{.} \end{array} \right\}$
- $g_d(s, \mathit{lessThan}(p, E)) = \left\{ \begin{array}{l} s\ \texttt{a sh:PropertyShape ;} \\ \quad \texttt{sh:path}\ b_E\ \ \texttt{;} \\ \quad \texttt{sh:lessThan}\ p\ \ \texttt{.} \end{array} \right\} \cup g_p(b_E, E)$
- $g_d(s, \mathit{lessThanEq}(p, E)) = \left\{ \begin{array}{l} s\ \texttt{a sh:PropertyShape ;} \\ \quad \texttt{sh:path}\ b_E\ \ \texttt{;} \\ \quad \texttt{sh:lessThanEq}\ p\ \ \texttt{.} \end{array} \right\} \cup g_p(b_E, E)$
- $g_d(s, \mathit{uniqueLang}(E)) = \left\{ \begin{array}{l} s\ \texttt{a sh:PropertyShape ;} \\ \quad \texttt{sh:path}\ b_E\ \ \texttt{;} \\ \quad \texttt{sh:uniquelang true .} \end{array} \right\} \cup g_p(b_E, E)$
- $g_d(s, \mathit{test}(t))$ is dependent on the test that t represents.

3
Recursion

In Chapter 1, we went over some of the ideas for what recursion for SHACL should look like, based on different semantics from the literature [6,28,30]. In this chapter, we formalize these ideas within the framework laid out in Chapter 2 and compare them with established notions of recursion from the knowledge representation literature.

We put forward a principled way to define semantics of recursive SHACL, building on *Approximation Fixpoint Theory* (AFT), an abstract lattice-theoretic framework originally designed to unify semantics of non-monotonic logics [34] with applications, among others, in logic programming, autoepistemic logic, default logic, abstract argumentation, and active integrity constraints [16,26,35,62,72,90].

There are several advantages to defining semantics of SHACL in this way:

- It is *simple* and *straightforward*: the power of AFT, comes largely from the fact that all that is required to apply it, is to define a (three-valued) semantic operator (similar to Fitting's immediate consequence operator for logic programs [42]). In many domains (including SHACL), there is a natural choice for such an operator; AFT then immediately induces all major classes of semantics.
- It provides *confidence*: AFT guarantees that the developed semantics follow well-established principles in non-monotonic reasoning, rather than ad-hoc solutions in which one can encounter issues that have been solved many times before, for instance that stable semantics respect principles such as *groundedness* [22]. All the application-specific work that needs to be done is to define a suitable lattice of "interpretations", and a suitable three-valued operator on this lattice. Defining such an operator is often significantly easier than directly defining a semantics. Even in case semantics are already defined, applying AFT can be a *sanity check*.
- It provides access to a *large body of theoretical results*, including theorems on *stratification* [17,99], *predicate introduction* [100], and *strong equivalence* [94], thereby eliminating the need to "reinvent the wheel" by rediscovering these results for SHACL.

In a nutshell, our main contribution is establishing formal foundations for the study of recursive SHACL.

3.1 Preliminaries: Approximation Fixpoint Theory

A *complete lattice* $\langle L, \leq \rangle$ is a set L equipped with a partial order $\leq$, such that every set $S \subseteq L$ has a least upper bound and a greatest lower bound. A complete lattice also has a least element $\perp$ and a greatest element $\top$. A function $O : L \to L$ will be called an *operator* here. We say O is $\leq$*-monotone* if $x \leq y$ implies that $O(x) \leq O(y)$. An element $x \in L$ is a *fixpoint* of O if $O(x) = x$. Every monotone operator O in a complete lattice has a least fixpoint, denoted $\mathrm{lfp}(O)$.

Given a lattice L, AFT uses a bilattice L^2. We define *projections* for pairs $(x, y) \in L^2$ as: $(x, y)_1 = x$ and $(x, y)_2 = y$. These pairs are used to approximate elements in the interval $[x, y] = \{z \mid x \leq z \wedge z \leq y\}$. We call $(x, y) \in L^2$ *consistent* if $x \leq y$, that is, if $[x, y]$ is non-empty. We use L^c to denote the set of consistent elements.

The *precision order* on L^2 is defined as $(x, y) \leq_p (u, v)$ if $x \leq u$ and $v \leq y$. If (u, v) is consistent, this means that (x, y) approximates all elements approximated by (u, v).

In its original form, AFT makes use of *approximators*, which are operators on L^2, but [36] showed that all the consistent fixpoints studied in AFT are uniquely determined by an approximator's restriction to L^c and developed a theory of *consistent approximators*.

An operator $A : L^c \to L^c$ is a *consistent approximator* of O if it is $\leq_p$-monotone and *coincides with O on L*, meaning $A(x, x) = (O(x), O(x))$ for all $x \in L$.

AFT studies fixpoints of O using fixpoints of A. Some fixpoints of interest are the following:

- The *A-Kripke-Kleene fixpoint* is the $\leq_p$-least fixpoint of A; it approximates all fixpoints of O.

- A *partial A-stable fixpoint* is a pair (x, y) such that $x = \mathrm{lfp}(A(\cdot, y)_1)$ and $y = \mathrm{lfp}(A(x, \cdot)_2)$, where $A(\cdot, y)_1 : L \to L$ maps z to $A(z, y)_1$ and similarly for $A(x, \cdot)_2$.

- The *A-well-founded fixpoint* is the least precise ($\leq_p$-least) partial A-stable fixpoint.

- An *A-stable fixpoint* of O is a fixpoint x of O such that (x, x) is a partial A-stable fixpoint.

These definitions allow reconstructing all major equally-named logic programming semantics by taking for O the immediate consequence operator T_P from van Emden and Kowalski [96]; for A we take Fitting's three- (or four-) valued extension Ψ_P [42].

3.2 Fixpoint Semantics for Recursive SHACL

For the rest of this Chapter, we use the formalization from Section 2.1 and fix a SHACL schema $\mathcal{S} = (D, T)$ and a graph-interpretation I.

In Section 2.1, we mentioned that if D is non-recursive, it uniquely induces a complete interpretation $I' := I \diamond D$ in which all constraints in T are to be verified. When D is recursive, however, the situation becomes more complex. On the one hand, there is a range of possible semantics dealing with recursion. On the other hand, some of the semantics yield not a single interpretation I', but either a set of them, or a *three-valued* interpretation. This will give us a choice between *brave* and *cautious* validation; the focus of this chapter is on the treatment of negation, but we briefly discuss brave and cautious validation below.

To apply AFT, the first step is to determine a suitable lattice. In our case, the obvious candidate is the lattice L_I (from now on, denoted L) of all interpretations I' with domain Δ^I that agree with I on $N \cup P$, or in other words, the set of interpretations that expand I. This set is equipped with the standard truth order, $I_1 \leq_t I_2$ if $[\![s]\!]^{I_1} \subseteq [\![s]\!]^{I_2}$ for all $s \in S$.

Next, we need a semantic operator T_D. The role of the semantic operator is to update the value of the interpretation of the shapes. In analogy with logic programming, its definition is straightforward: it maps the interpretation I' to $T_D(I')$ such that for each shape name s with defining rule $s \leftarrow \varphi$, we define the truth value for s in I' as $T_D(I')(s) = [\![\varphi]\!]^{I'}$.

Table 3.1: Three-valued semantics of shapes.

φ	$\llbracket\varphi\rrbracket^{\mathcal{I}}(a)$
$\top$	$\mathbf{t}$
$\mathit{hasValue}(c)$	$\begin{cases} \mathbf{t} & \text{if } a = c \\ \mathbf{f} & \text{otherwise} \end{cases}$
$\mathit{hasShape}(s)$	$\llbracket s\rrbracket^{\mathcal{I}}(a)$
$\neg\varphi_1$	$\neg\llbracket\varphi_1\rrbracket^{\mathcal{I}}(a)$
$\varphi_1 \wedge \varphi_2$	$\min_{\leq_t}(\llbracket\varphi_1\rrbracket^{\mathcal{I}}(a), \llbracket\varphi_2\rrbracket^{\mathcal{I}}(a))$
$\varphi_1 \vee \varphi_2$	$\max_{\leq_t}(\llbracket\varphi_1\rrbracket^{\mathcal{I}}(a), \llbracket\varphi_2\rrbracket^{\mathcal{I}}(a))$
$\geq_n E.\varphi_1$	$\begin{cases} \mathbf{t} & \text{if } \sharp\{b \in \llbracket E\rrbracket^{\mathcal{I}}(a) \mid \llbracket\varphi_1\rrbracket^{\mathcal{I}}(b) = \mathbf{t}\} \geq n, \\ \mathbf{f} & \text{if } \sharp\{b \in \llbracket E\rrbracket^{\mathcal{I}}(a) \mid \llbracket\varphi_1\rrbracket^{\mathcal{I}}(b) \geq_t \mathbf{u}\} < n, \\ \mathbf{u} & \text{otherwise} \end{cases}$
$\mathit{eq}(p, E)$	$\begin{cases} \mathbf{t} & \text{if } a \in \llbracket \mathit{eq}(p, E)\rrbracket^{\mathcal{I}} \\ \mathbf{f} & \text{otherwise} \end{cases}$
$\mathit{disj}(p, E)$	$\begin{cases} \mathbf{t} & \text{if } a \in \llbracket \mathit{disj}(p, E)\rrbracket^{\mathcal{I}} \\ \mathbf{f} & \text{otherwise} \end{cases}$
$\mathit{closed}(Q)$	$\begin{cases} \mathbf{t} & \text{if } \llbracket p\rrbracket^{\mathcal{I}}(a) = \emptyset \text{ for all } p \in P \setminus Q \\ \mathbf{f} & \text{otherwise} \end{cases}$

With the lattice $\langle L, \leq_t\rangle$, elements of L^c are pairs $\mathcal{I} = (I_1, I_2)$ of two interpretations with $I_1 \leq_t I_2$; such pairs correspond one-to-one to *three-valued interpretations* that assign each $s \in S$ a function $\Delta \to \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$, mapping a to $\mathbf{t}$ if a in $\llbracket s\rrbracket^{I_1}$, to $\mathbf{f}$ if $a \notin \llbracket s\rrbracket^{I_2}$ and to $\mathbf{u}$ otherwise (in other words, I_1 represents what is *certainly true* and I_2 what is *possibly true*). From now on, we simply refer to elements of L^c as three-valued interpretations.

We can evaluate a shape φ in a three-valued interpretation $\mathcal{I}$ with a straightforward extension of Kleene's truth tables, as also used in previous studies of recursive SHACL [6, 30]. For a node $a \in N$, we write $\llbracket\varphi\rrbracket^{\mathcal{I}}(a)$ to denote three valued evaluation of a for φ in $\mathcal{I}$. Table 3.1 gives the three-valued evaluation of a shape. This table makes use of the truth order $\leq_t$ on truth values defined as $\mathbf{f} \leq_t \mathbf{u} \leq_t \mathbf{t}$, and the negation on truth values defined as usual: $\neg\mathbf{t} = \mathbf{f}; \neg\mathbf{f} = \mathbf{t}; \neg\mathbf{u} = \mathbf{u}$.

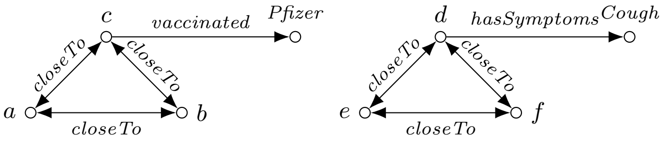
Once a three-valued evaluation of shapes is defined, an approximator is obtained directly. Like the operator, the approximator updates the value of each shape symbol according to its defining rule: it maps $\mathcal{I}$ to $\Psi_D(\mathcal{I})$ where for each shape $s \in S$ defined by the rule $s \leftarrow \varphi$, $\Psi_D(\mathcal{I})(s) = \llbracket\varphi\rrbracket^{\mathcal{I}}$.

Theorem 3.1. *Ψ_D is a consistent approximator of T_D.*

Proof. The fact that Ψ_D is $\leq_p$-monotonic follows directy from the fact that the three-valued truth evaluation is $\leq_p$-monotonic: given two three-valued interpretations $\mathcal{I}$ and $\mathcal{I}'$, when $\mathcal{I} \leq_p \mathcal{I}'$, it is easily verified using the definition from Table 3.1 that $\Psi_D(\mathcal{I}) \leq_p \Psi_D(\mathcal{I}')$. Furthermore, it is easy to verify that on $\mathcal{I} = (I_1, I_2)$ with $I_1 = I_2$, the two-valued evaluation of Table 2.2 and the three-valued evaluation of Table 3.1 coincide. From that it follows that Ψ_D indeed agrees with T_D on L. □

At this point, AFT dictates what the supported models (fixpoints of T_D), (partial) stable models (Ψ_D-stable fixpoints), well-founded model (Ψ_D-well-founded fixpoint), and Kripke-Kleene model (Ψ_D-KK fixpoint) of D are. It is worth stressing that to arrive to this point, we made two choices:

Figure 3.1: Visual representation of the example interpretation.



- The first was our choice of order on the lattice. We opted here for the truth order, but its inverse would also have been a possible choice. Several of the semantics induced by AFT aim to minimize models in the chosen order for reasons of *groundedness* [22], e.g., if s has $s \leftarrow s$ as defining rule, in stable and well-founded semantics, our chosen order would result in no nodes satisfying s.

- The second choice we made is *which three-valued truth evaluation to use*; we opted for the most obvious choice: a direct extension of Kleene's three-valued truth tables, which was used in other studies of recursive SHACL as well [6, 30].

Given these choices, models of the different types, like stable- and well-founded models, are defined by AFT, and hence semantics for brave and cautious validation under of each semantics are established as well.

Definition 3.2. Let $\sigma \in \{KK, WF\}$ and let $\mathcal{I}$ be the σ-model. We say that I *cautiously (resp. bravely) σ-validates* with respect to (D, T) if $[\![\varphi \wedge \neg hasShape(s)]\!]^{\mathcal{I}} = \mathbf{f}$ (resp. $[\![\varphi \wedge \neg hasShape(s)]\!]^{\mathcal{I}} \leq_t \mathbf{u}$) for every $\varphi \subseteq s$ in T.

Let $\sigma \in \{St, Sup\}$ and let M be the set of σ-models. We say that I *cautiously (resp. bravely) σ-validates* with respect to (D, T) if $[\![\varphi \wedge \neg hasShape(s)]\!]^{I'} = \mathbf{f}$ for all (resp. for some) $I' \in M$ for every $\varphi \subseteq s$ in T.

Let us illustrate the differences between the various types of models on a small example.

Example 3.3. Consider binary predicates *closeTo*, *hasSymptoms*, and *vaccinated* and an interpretation I with domain $\{a, b, c, d, e, f, \mathit{Pfizer}, \mathit{Cough}\}$, where $a, \ldots, f$ represent people (divided in two cliques of three "close" friends); one person (c) is vaccinated and one person (d) shows Covid symptoms. This interpretation is visually depicted in 3.1.

We define two shapes: the shape of people at risk (those who (1) are not vaccinated and (2) have symptoms or are close to someone at risk) and the shape of people who can go to office (those who are not at risk), as formalized below:

$$\begin{aligned} \mathit{atRisk} &\leftarrow \neg\exists \mathit{vaccinated}.\top \wedge (\exists \mathit{hasSymptoms}.\top \vee \exists \mathit{closeTo}.\mathit{hasShape}(\mathit{atRisk})) \\ \mathit{canWork} &\leftarrow \neg \mathit{hasShape}(\mathit{atRisk}) \end{aligned}$$

These shape definitions can be written in real SHACL, as shown in Listing 3.1. For this set of shape definitions, the unique stable model equals the well-founded model and states that d, e, and f are at risk, while a, b, and c can work. In the Kripke-Kleene model, d, e, and f are again at risk, c is not at risk (and hence can work), but for a and b it is unknown whether they are at risk. There are two supported models: the stable model and one in which everyone except for c is at risk.

3.3 Comparison with Existing Semantics

Corman et al. [30] already defined a supported semantics, Andreşel et al. [6] a stable semantics, and later Chmuroviĉ and Šimkus [28] defined the well-founded semantics directly for

Listing 3.1: The *atRist* and *canWork* shapes from Example 3.3 written in real SHACL

```
:atRisk a sh:NodeShape ;
  sh:not [
    sh:path :vaccinated ;
    sh:minCount 1 ;
  ] ;
  sh:or (
    [ sh:path :hasSymptoms ;
      sh:minCount 1 ;
    ]
    [ sh:path :closeTo ;
      sh:qualifiedValueShape :atRisk ;
      sh:qualifiedMinCount 1 ;
    ]
  ) .

:canWork a sh:NodeShape ;
  sh:not :atRisk .
```

SHACL. For clarity, we refer to the first two semantics as CRS-supported, and ACORSS-stable semantics respectively, and to the semantics induced by AFT, e.g., as AFT-stable. Both Corman et al. and Andresel et al. focus on brave validation, but Andreşel et al. also mention the possibility of cautious validation. The main results on correspondence between the semantics are summarized in this Section.

Interestingly, Corman et al. [30, Definition 5] already defined the three-valued immediate consequence operator Ψ_D (there denoted $\mathbf{T}$). While the focus of that work was on supported semantics, we now showed that in fact, by defining the approximator Ψ_D, they had everything at hand to define the full. Since Corman et al. also characterized supported models as fixpoints of Ψ_D (in their Definition 17), our semantics and theirs coincide.

Theorem 3.4. *I CRS-validates (D, T) if and only if I bravely Sup-validates (D, T).*

Proof. This follows directly from the fact Corman et al. [30] also defined an "Immediate evaluation operator T", which coincides with our operator T_D and their supported models (there called *faithful assignments*), are simply fixpoints of T. □

The situation is somewhat different for stable semantics, which Andreşel et al. [6] defined in terms of level mappings.

Definition 3.5 ([6])**.** Let I' be an interpretation. A *level assignment* for I' is a function *level* that maps tuples in $\{(\varphi, a) \mid \llbracket\varphi\rrbracket^{I'}(a) = \mathbf{t}\}$ to integers and satisfies

1. $\mathit{level}(\varphi_1 \wedge \varphi_2, a) = \max\{\mathit{level}(\varphi_1, a), \mathit{level}(\varphi_2, a)\}$,
2. $\mathit{level}(\varphi_1 \vee \varphi_2, a) = \min\{\mathit{level}(\varphi_i, a) \mid i \in \{1, 2\} \wedge \llbracket\varphi_i\rrbracket^{I'}(a) = \mathbf{t}\}$,
3. $\mathit{level}(\geq_n E.\varphi)$ is the smallest $k \geq 0$ for which there are n elements $b_1, \ldots b_n \in \Delta^{I'}$ such that $\mathit{level}(\varphi, b_i) \leq k$, $(a, b_i) \in \llbracket E\rrbracket^{I'}$, and $\llbracket\varphi\rrbracket^{I'}(b_i) = \mathbf{t}$, and
4. $\mathit{level}(\forall E.\varphi, a) = \max(\{\mathit{level}(\varphi, b) \mid (a, b) \in \llbracket E\rrbracket^{I'} \wedge \llbracket\varphi\rrbracket^{I'}(b) = \mathbf{t}\}$.

A supported model I' is an ARCOSS-stable model if there exists a level assignment for I' such that $\mathit{level}(s, a) > \mathit{level}(\varphi, a)$ for each rule $s \leftarrow \varphi$ in D and each a with $\llbracket s\rrbracket^{I'}(a) = \mathbf{t}$.

We recall the definition of “shape normal form” of Andreşel et al. [6]:

Definition 3.6. D is in *shape normal form* if all rules in D have one of the following forms:

$$\begin{array}{lll} s \leftarrow \top & s \leftarrow \mathit{hasValue}(c) & s \leftarrow \mathit{hasShape}(s') \wedge \mathit{hasShape}(s'') \\ s \leftarrow \mathit{disj}(p, E') & s \leftarrow \neg \mathit{hasShape}(s') & s \leftarrow \mathit{hasShape}(s') \vee \mathit{hasShape}(s'') \\ s \leftarrow \mathit{eq}(p, E') & s \leftarrow \forall E.\mathit{hasShape}(s') & s \leftarrow \geq_n E.\mathit{hasShape}(s') \end{array}$$

Theorem 3.7. *If I' is an AFT-stable model of D, then it is also an ACORSS-stable model. If D is in shape normal form, the converse also holds.*

We will show both directions separately in the following propositions.

Proposition 3.8. *Let D be a set of shape definitions. Let M be a (two-valued) Σ-interpretation. If M is an Ψ_D-stable model, then M is also an ACORSS-stable model for D.*

All interpretations are assumed to agree on $\Sigma \cap (N \cup P)$, i.e., the only differ on the shape symbols. When constructing interpretations, their value of the agreed upon symbols will not be mentioned explicitly here.

Proof. We define the sequence of (two-valued) Σ-interpretations $(I_i)_{i\in\mathbb{N}}$ with $n \in \mathbb{N}$ as follows:

- $I_0 =$ the interpretation such that $[\![s]\!]^{I_0} = \emptyset$ for all $s \in S$,
- $I_i = \Psi_D(I_{i-1}, M)_1$,
- $I_n = \text{lfp}(\Psi_D(\cdot, M)_1)$.

Even though there are an infinite number of possible interpretations, our sequence is always finite. This is because there is a finite number of elements from the domain that are relevant to the interpretation (the ones actually used in the interpretation of predicates, and the ones mentioned in the shape). As Ψ_D is $\leq_p$-monotone, and there are a finite number of possible interpretations, this sequence will reach a fixed-point.

We construct a level mapping *level* as follows.

- For all formulas φ of the form $\top$, $\mathit{hasValue}(c)$, $\neg\psi$, $\mathit{eq}(p, E)$ or $\mathit{disj}(p, E)$, if $a \in [\![\varphi]\!]^M$, then $\mathit{level}(\varphi, a) = 0$.
- Next, for every $a \in [\![s]\!]^{I_i}$: $\mathit{level}(s, a)$ is the smallest j such that $a \in [\![s]\!]^{I_j}$.
- Now, the rest of the definition of *level* follows from these assignments together with the definition of a level assignment.

To show that M is indeed an ACORSS-stable model, we show that if $\mathit{level}(\varphi, a) = i$, then $\mathit{level}(s, a) \geq i + 1$ with $s \leftarrow \varphi \in D$.

Before we can prove this claim, we show that for all $i \in \mathbb{N}$: $[\![\varphi]\!]^{(I_i,M)}(a) = \mathbf{t} \iff \mathit{level}(\varphi, a) \leq i$.

This is proven by induction on the structure of φ. First, consider the case for φ is $\top$, $\mathit{hasValue}(c)$, $\neg\psi$, $\mathit{eq}(p, E)$ or $\mathit{disj}(p, E)$. Since all considered interpretations agree on constants and predicate symbols, this means that if $[\![\varphi]\!]^{(I_i,M)}(a) = \mathbf{t}$, then $\mathit{level}(\varphi, a) \leq i$ as $\mathit{level}(\varphi, a) = 0$. The converse clearly holds as well. We consider the following inductive cases:

When φ is of the form $\mathit{hasShape}(s)$, assume $[\![s]\!]^{(I_i,M)}(a) = \mathbf{t}$. Then, we have defined $\mathit{level}(s, a)$ to be the smallest j such that $[\![s]\!]^{I_j}(a) = \mathbf{t}$, clearly $j \leq i$, therefore $\mathit{level}(s, a) \leq i$. For the other direction assume $\mathit{level}(s, a) \leq i$, therefore, by our definition from before and the definition of the sequence $(I_i)_{i\in\mathbb{N}}$, $[\![s]\!]^{I_i}(a) = \mathbf{t}$.

When φ is of the form $\varphi_1 \wedge \varphi_2$, assume $[\![\varphi]\!]^{(I_i,M)}(a) = \mathbf{t}$. Then, $[\![\varphi_1]\!]^{(I_i,M)}(a) = \mathbf{t}$ and $[\![\varphi_2]\!]^{(I_i,M)}(a) = \mathbf{t}$. By induction, $\mathit{level}(\varphi_1, a) \leq i$ and $\mathit{level}(\varphi_2, a) \leq i$. By definition of a level assignment, $\mathit{level}(\varphi, a) = \mathit{level}(\varphi_1 \wedge \varphi_2, a) = \max(\mathit{level}(\varphi_1, a), \mathit{level}(\varphi_2, a)) \leq i$. For the other direction, assume $\mathit{level}(\varphi, a) \leq i$, therefore $\mathit{level}(\varphi_1, a) \leq i$ and $\mathit{level}(\varphi_2, a) \leq i$. By induction, $[\![\varphi_1]\!]^{(I_i,M)}(a) = \mathbf{t}$ and $[\![\varphi_2]\!]^{(I_i,M)}(a) = \mathbf{t}$. By definition of three-valued evaluation $[\![\varphi]\!]^{(I_i,M)}(a) = \mathbf{t}$.

When φ is of the form $\varphi_1 \vee \varphi_2$, assume $[\![\varphi]\!]^{(I_i,M)}(a) = \mathbf{t}$. Then at least $[\![\varphi_1]\!]^{(I_i,M)}(a) = \mathbf{t}$ or $[\![\varphi_2]\!]^{(I_i,M)}(a) = \mathbf{t}$. Assume w.l.o.g. $[\![\varphi_1]\!]^{(I_i,M)}(a) = \mathbf{t}$. Then, by induction $\mathit{level}(\varphi_1, a) \leq i$, so by definition of a level assignment $\mathit{level}(\varphi_1, a) \leq i$. For the other direction, assume $\mathit{level}(\varphi, a) \leq i$, therefore at least one of $\mathit{level}(\varphi_1, a) \leq i$ or $\mathit{level}(\varphi_2, a) \leq i$. Thus, by induction, $[\![\varphi_1 \vee \varphi_2]\!]^{(I_i,M)}(a) = \mathbf{t}$.

When φ is $\geq_n E.\varphi_1$, assume $[\![\varphi]\!]^{(I_i,M)}(a) = \mathbf{t}$. Thus, there exist n elements $b_1, \ldots, b_n$ such that $(a, b_k) \in [\![E]\!]^M$, furthermore $[\![\varphi_1]\!]^{(I_i,M)}(b_k) = \mathbf{t}$ with $1 \leq k \leq n$. By induction $\mathit{level}(\varphi_1, b_k) \leq i$. By definition of level assignments, $\mathit{level}(\varphi, a) \leq i$. For the other direction, assume $\mathit{level}(\varphi, a) \leq i$, therefore there exist n elements $b_1, \ldots, b_k$ such that $(a, b_k) \in [\![E]\!]^M$ and $\mathit{level}(\varphi_1, b_k) \leq i$. By induction $[\![\varphi_1]\!]^{(I_i,M)}(b_k) = \mathbf{t}$ and thus, by definition of three-valued evaluation, $[\![\varphi]\!]^{(I_i,M)}(a) = \mathbf{t}$.

Finally, now we can show our Proposition. For every shape rule $s \leftarrow \varphi$. We know that $\mathit{level}(s, a)$ is the smallest i such that $a \in [\![s]\!]^{I_i}$. By the definition of the sequence $(I_i)_{i \in \mathbb{N}}$ we have $[\![\varphi]\!]^{(I_{i-1},M)}(a) = \mathbf{t}$. Given our claim proven above, $\mathit{level}(\varphi, a) \leq i - 1 < i = \mathit{level}(s, a)$. □

Proposition 3.9. *Let D be a set of shape definitions in normal form. Let M be a (two-valued) Σ-interpretation. If M is an ACORSS-stable model for D, then M is also a Ψ_D-stable model.*

Proof. Let *level* be the level mapping for the ACORSS-stable model M. Let m be the maximal level mapping level from *level*. We define the sequence $(I_i)_{i \leq m}$ of Σ-interpretations as follows:

$$[\![s]\!]^{I_i} = \{a \mid \mathit{level}(s, a) \leq i\} \text{ for all } s \in S,\ i \leq m$$

In particular, it holds that $[\![s]\!]^{I_0} = \emptyset$ for all s and that $I_m = M$ (by the definition of a level mapping). We also define a sequence of S-interpretations $(J_i)_{i \leq m}$ as follows:

- $J_0 =$ the interpretation such that $[\![s]\!]^{J_0} = \emptyset$ for all $s \in S$,
- $J_i = \Psi_D(J_{i-1}, M)_1$,
- $J_\infty = \mathrm{lfp}(\Psi_D(\cdot, M)_1)$.

Even though there are an infinite number of possible interpretations, our sequence is always finite. This is because there is a finite number of elements from the domain that are relevant to the interpretation (the ones actually used in the interpretation of predicates, and the ones mentioned in the shape). As Ψ_D is $\leq_p$-monotone, and there are a finite number of possible interpretations, this sequence will reach a fixed-point.

We show that for every $0 \leq i \leq m : I_i \subseteq J_i$. The claim holds for $I_0 = J_0 = \emptyset$. Assume the claim holds for $i - 1$, we show the claim holds for i. Let $s(a) \in I_i$ and $s \leftarrow \varphi$, and note that $\mathit{level}(s, a) > \mathit{level}(\varphi, a)$. We show $I_i \subseteq J_i$ through the following cases for φ.

When φ is of the form $\top, \mathit{hasValue}(c), \mathit{eq}(p, E)$ or $\mathit{disj}(p, E)$: since the truth value does not depend on the S-interpretations and $s(a) \in I_i$, we know that $s(a) \in J_i$.

When φ is of the form $\neg \mathit{hasShape}(s_1)$: as $s(a) \in I_i$, we have $s_1(a) \notin I_i$ therefore, by definition of three-valued evaluation, $s_1(S) \notin J_i$ and $s(a) \in J_i$.

When φ is of the form $\mathit{hasShape}(s_1) \wedge \mathit{hasShape}(s_2)$: by definition of ACORSS-stable models: $i > \max(\mathit{level}(s_1, a), \mathit{level}(s_2, a))$. Therefore, by induction $s_1(a), s_2(a) \in J_{i-1}$. By the definition of three-valued truth evaluation we have $s(a) \in J_i$.

Listing 3.2: The *safe* shape from Example 3.10 written in real SHACL

```
:safe a sh:NodeShape ;
  sh:or (
    [ sh:path :vaccinated ;
      sh:minCount 1 ;
    ]
    [ sh:path :closeTo ;
      sh:qualifiedValueShape [ sh:not :safe ] ;
      sh:qualifiedMaxCount 1 ;
    ]
  ) .
```

When φ is of the form $\mathit{hasShape}(s_1) \vee \mathit{hasShape}(s_2)$: we know by definition of ACORSS-stable models: $i > \min(\mathit{level}(s_1, a), \mathit{level}(s_2, a))$. Therefore, by induction at least one of $s_1(a), s_2(a)$ is in J_{i-1}. By the definition of three-valued evaluation we have $s(a) \in J_i$.

When φ is of the form $\geq_n E.\mathit{hasShape}(s_1)$: we know by definition that there are n nodes $b_1, \dots, b_n$ such that for all $1 \leq j \leq n : (a, b_j) \in [\![E]\!]^M$ Also, by definition of the level mapping: for every b_j there exists an $m < i$ such that $s_1(b_j) \in I_m$. By induction $s_1(b_j) \in J_m$ and thus $s(a) \in J_i$.

From the claim follows that $M = I_m \subseteq J_m$. By the construction of J_m, we have $J_m \leq \mathrm{lfp}(\Psi_D(\cdot, I))_1$. Now since M is a fixpoint of this operator, it must be that $I_m = M$ and hence that M is an Ψ_D-stable model. □

The difference between our stable semantics and the ACORSS-stable semantics is a *semantic* (in terms of the standard three-valued truth evaluation) versus a *syntactic* (the level mappings are defined in terms of the syntactic structure of the shapes) treatment of negation and is illustrated in the next example.

Example 3.10 (Example 3.3 continued)**.** Suppose that in our same interpretation, we wish to define a shape that identifies possible superspreaders. To do this, we say that a person is "safe" if they are vaccinated, or in contact with at most 1 non-safe person. This can be formalized as:

$$\mathit{safe} \leftarrow \exists \mathit{vaccinated}.\top \vee \ \leq_1 \mathit{closeTo}.\neg \mathit{hasShape}(\mathit{safe}),$$

Where $\leq_1$ is an abbreviation for $\neg \geq_2$. This shape definition can be written in real SHACL, as shown in Listing 3.2. With the interpretation described in Example 3.3, there is a single AFT-stable model in which a, b, and c are safe, but d, e, and f are not.

However, there are two ACROSS-stable models: the one mentioned above, and one in which everyone is safe, including the three-clique of non-vaccinated people.

Finally, the work by Chmuroviĉ and Šimkus [28] proposes the Well-founded semantics for SHACL. Their definition corresponds closely to the definition for logic programs due to Van Gelder et al. [97], which is known to correspond to the Well-founded semantics given through AFT. To see their semantics corresponds to ours, we only need to verify their three-valued evaluation of shape expressions corresponds to ours, which is the case.

4 Expressiveness

When a complicated but influential new tool is proposed in the community, in our case SHACL, we feel it is important to have a solid understanding of its design. The task we are focusing on in this chapter is checking conformance of RDF graphs against shape schemas. Every shape schema $\mathcal{S}$ defines a decision problem: given an RDF graph G, check whether G conforms to $\mathcal{S}$. In database terms, we are processing a boolean query on a graph database. In description logic terms, this amounts to model checking of a TBox: given an interpretation, check whether it satisfies the TBox.

Some of the constructs from SHACL are well known concept constructors from expressive description logics [25]: the boolean connectives; constants; qualified number restriction (a combination of existential quantification and counting); and regular path expressions with inverse. However, SHACL also has three specific logical features that are less common: equality, disjointness, and closure constraints.

Our goal in this Chapter is to clarify the impact of these uncommon features of SHACL on its expressiveness as a language for boolean queries on graph databases. Thereto, we offer the following contributions.

- We show that each of the three features is primitive in a strong sense. Specifically, for each feature, we exhibit a boolean query Q such that Q is expressible by a single target–shape pair, using only the feature and the basic constructs; however, Q is not expressible by any generalized shape schema when the feature is disallowed.

- We also clarify the significance of the restriction that SHACL puts on allowed targets. We observe that as long as closure constraints are not used, the restriction is actually insignificant. Any generalized shape schema, allowing arbitrary but closure-free shapes on the left-hand sides of the inclusion statements, can be equivalently written as a shape schema with only targets on the left-hand sides. However, allowing closure constraints on the left-hand side strictly adds expressive power.

- We additionally show that “full” variants of equality tests or disjointness tests result in strictly more expressive languages. This result anticipates planned extensions of SHACL [56].

- Our results continue to hold when the definition of *recursive* shapes is allowed, provided that recursion through negation is stratified.

This Chapter is organized as follows. Section 4.1 and Section 4.2 present our results, and Section 4.3 extends our result for "full" equality and disjointness tests. Section 4.4 presents the extension to stratified recursion.

4.1 Expressiveness of SHACL features

Throughout this Chapter, we use the formalization of the logical core of SHACL described in Section 2.1, except, the treatment here omits shape names. Shape names are redundant as far as expressive power is concerned, as long as we are in a non-recursive setting, because shape definitions can then always be unfolded. Indeed, for clarity of exposition, we have chosen to work first with non-recursive shape schemas. Section 4.4 then presents the extension to (stratified) recursion (and introduces shape names in the process).

We now recall the definition of a *generalized shape schema* (or shape schema for short) as a finite set of inclusion statements, where an inclusion statement is of the form $\varphi_1 \subseteq \varphi_2$, with φ_1 and φ_2 shapes. A *target-based shape schema* is a shape schema that only uses targets allowed by real SHACL on the left-hand sides of its inclusion statements.

Recall that a graph G *conforms* to a shape schema $\mathcal{S}$, denoted by $G \models \mathcal{S}$, if $[\![\varphi_1]\!]^G$ is a subset of $[\![\varphi_2]\!]^G$, for every statement $\varphi_1 \subseteq \varphi_2$ in $\mathcal{S}$.

Thus, any shape schema $\mathcal{S}$ defines the class of graphs that conform to it. We denote this class of graphs by

$$[\![\mathcal{S}]\!] := \{\text{graph } G \mid G \models \mathcal{S}\}.$$

Accordingly, two shape schemas $\mathcal{S}_1$ and $\mathcal{S}_2$ are said to be *equivalent* if $[\![\mathcal{S}_1]\!] = [\![\mathcal{S}_2]\!]$.

Methodology. A *feature set* F is a subset of $\{\mathit{eq}, \mathit{disj}, \mathit{closed}\}$. The set of all shape schemas using only features from F, besides the standard constructs, is denoted by $\mathcal{L}(F)$. In particular, shape schemas in $\mathcal{L}(\emptyset)$ use only the standard constructs and none of the three features. Specifically, they only involve shapes built from boolean connectives, constants, and qualified number restrictions, with path expressions built from property names, *id* and the standard operators union, composition, and Kleene star.

We say that feature set F_1 is *subsumed* by feature set F_2, denoted by $F_1 \preceq F_2$, if every shape schema in $\mathcal{L}(F_1)$ is equivalent to some shape schema in $\mathcal{L}(F_2)$. As it will turn out,

$$F_1 \preceq F_2 \quad \Leftrightarrow \quad F_1 \subseteq F_2, \tag{$*$}$$

or intuitively, "every feature counts." Note that the implication from right to left is trivial, but the other direction is by no means clear from the outset.

More specifically, for every feature, we introduce a class of graphs, as follows. In what follows we fix some property name r. We say a class of graphs is definable by $\mathcal{L}(F)$ if there exists a shape schema in $\mathcal{L}(F)$ that defines it.

Equality Q_{eq} is the class of graphs where all r-edges are symmetric. Note that Q_{eq} is definable in $\mathcal{L}(\mathit{eq})$ by the single, target-based, inclusion statement $\exists r.\top \subseteq \mathit{eq}(r^-, r)$.

Disjointness Q_{disj} is the class of graphs where all nodes with an outgoing r-edge have at least one symmetric r-edge. This time, Q_{disj} is definable in $\mathcal{L}(\mathit{disj})$, by the single, target-based, inclusion statement $\exists r.\top \subseteq \neg\mathit{disj}(r^-, r)$.

Closure Q_{closed} is the class of graphs where for all nodes with an outgoing r-edge, all outgoing edges have label r. Again Q_{closed} is definable in $\mathcal{L}(\mathit{closed})$ by the single, target-based, inclusion statement $\exists r.\top \subseteq \mathit{closed}(r)$.

Listing 4.1: The four shapes used to express the classes of graphs Q_{eq}, Q_{disj}, and Q_{closed} in real SHACL

```
:equality a sh:PropertyShape ;
  sh:targetObjectsOf :r ;
  sh:path [ sh:inversePath :r ] ;
  sh:equals :r .

:disjointness a sh:Nodeshape ;
  sh:targetObjectsOf :r ;
  sh:not [ sh:path [ sh:inversePath :r ] ;
           sh:disjoint :r ] .

:closedness a sh:NodeShape ;
  sh:targetObjectsOf :r ;
  sh:closed true ;
  sh:ignoredProperties ( :r ) .
```

To show the connection to real SHACL, the constraints described above can be written in real SHACL as shown in Listing 4.1.

We establish the following theorem, from which the above equivalence $(*)$ immediately follows:

Theorem 4.1. *Let $X \in \{eq, disj, closed\}$ and let F be a feature set with $X \notin F$. Then Q_X is not definable in $\mathcal{L}(F)$.*

For $X = closed$, Theorem 4.1 is proven differently than for the other two features. First, we deal with the remaining features through the following concrete result, illustrated in Figure 4.1. The formal definition of the graphs illustrated in Figure 4.1 for $X = disj$ will be provided in Definition 4.9.

Proposition 4.2. *Let $X = disj$ or eq. Let Σ be a finite vocabulary including r, and let m be a nonzero natural number. There exist two graphs G and G' with the following properties:*

1. *G' belongs to Q_X, but G does not.*
2. *For every shape φ over Σ such that φ does not use X, and φ counts to at most m, we have*
$$[\![\varphi]\!]^G = [\![\varphi]\!]^{G'}.$$

Here, "counting to at most m" means that all quantifiers $\geq_n$ used in φ satisfy $n \leq m$. For $X = eq$, this proposition is reformulated as Proposition 4.13, and for $X = disj$, this proposition is reformulated as Proposition 4.16.

To see that Proposition 4.2 indeed establishes Theorem 4.1 for the three features under consideration, we use the notion of *validation shape* of a shape schema. This shape evaluates to the set of all nodes that violate the schema. Thus, the validation shape is an abstraction of the "validation report" in SHACL [86]: a graph conforms to a schema if and only if the validation shape evaluates to the empty set. The validation shape can be formally constructed as the disjunction of $\varphi_1 \wedge \neg\varphi_2$ for all statements $\varphi_1 \subseteq \varphi_2$ in the schema.

Now consider a shape schema $\mathcal{S}$ not using feature X. Let m be the maximum count used in shapes in $\mathcal{S}$, and let Σ' be the set of constants and property names mentioned in $\mathcal{S}$. Now given $\Sigma = \Sigma' \cup \{r\}$ and m, let G and G' be the two graphs exhibited by the Proposition, and let φ be the validation shape for $\mathcal{S}$. Then φ will evaluate to the same result on G and

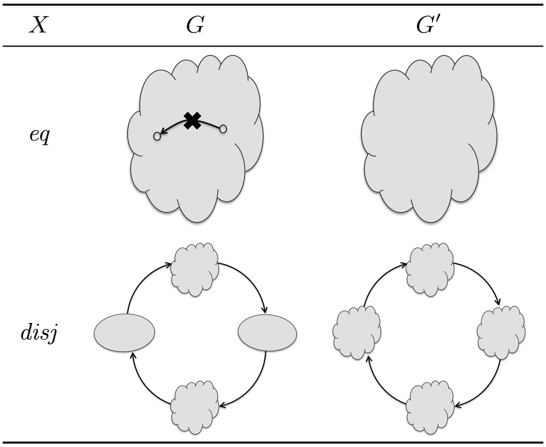


Figure 4.1: Graphs used to prove Proposition 4.2. The nodes are taken outside Σ. For $X = eq$, the cloud shown for G' represents a complete directed graph on $m + 1$ nodes, with self-loops, and G is the same graph with one directed edge removed. For $X = disj$, in the picture for G, each cloud again stands for a complete graph, but this time on $M = \max(m, 3)$ nodes, and without the self-loops. Each oval stands for a set of M separate nodes. An arrow from one blob to the next means that every node of the first blob has a directed edge to every node of the next blob. So, G is a directed 4-cycle of alternating clouds and ovals, and G' is a directed 4-cycle of clouds.

G'. However, for $\mathcal{S}$ to define Q_X, validation would have to return the empty set on G' but a nonempty set on G. We conclude that $\mathcal{S}$ does not define Q_X.

We will prove Proposition 4.2 for $X = disj$ in Section 4.1.2, and $X = eq$ in Section 4.1.3. We will show Theorem 4.1 for $X = closed$ in Section 4.1.4. However, we first need to establish some preliminaries on path expressions.

4.1.1 Preliminaries on path expressions

We call a path expression E *equivalent* to a path expression E' when for every graph G, $[\![E]\!]^G = [\![E']\!]^G$. We call a path expression E *id-free* whenever id is not present in the expression.

Lemma 4.3. *Every path expression E is equivalent to: id, or $E' \cup id$, or E' where E' is an id-free path expression.*

Proof. The proof is by induction on the structure of E. When E is id-free or id, the claim directly follows. We consider the following inductive cases:

- E is E_1/E_2. By induction, we consider nine cases. When both E_1 and E_2 are id-free, E is id-free. Whenever E_1 is id, clearly E is equivalent to E_2. Analogously, whenever E_2 is id, E is equivalent to E_1.

  Consider the two cases where E_1 is $E'_1 \cup id$ with E'_1 an id-free path expression. First, when E_2 is $E'_2 \cup id$ with E'_2 an id-free path expression, then E is equivalent to $E'_1/E'_2 \cup E'_1 \cup E'_2 \cup id$ which is of the form $E' \cup id$ with E' the id-free path expression $E'_1/E'_2 \cup E'_1 \cup E'_2$. Second, when E_2 is id-free, E is equivalent to $E'_1/E_2 \cup E_2$ which is id-free.

  Finally, consider the two case where E_1 is id-free and E_2 is $E'_2 \cup id$ with E'_2 an id-free path expression, then E is equivalent to $E_1/E'_2 \cup E_1$ which is id-free.

- E is $E_1 \cup E_2$. This case follows immediately by induction.

- E is E_1^*. There are three cases. When E_1 is id, E is equivalent to id. When E_1 is $E_1' \cup id$ with E_1' an id-free path expression, then E is equivalent to $E_1'^*$ and clearly $E_1'^*$ is id-free. Lastly, when E_1 is id-free, clearly E is as well. □

We also need the notion of "safe" path expressions together with the following Lemma, detailing how path expressions can behave on the nodes outside a graph. One can divide all path expressions into the "safe" and the "unsafe" ones.

Definition 4.4 (Safety)**.** A path expression is *safe* if one of the following conditions holds:

- E is p or p^- with p a property name

- E is $E_1 \cup E_2$ and both E_1 and E_2 are safe

- E is E_1/E_2 and at least one of E_1 or E_2 is safe

Otherwise, E is *unsafe*.

Lemma 4.5. *Let E be an id-free path expression and let G be a graph.*

- *If E is safe, then $[\![E]\!]^G \subseteq N_G \times N_G$.*

- *If E is unsafe, then $[\![E]\!]^G = ([\![E]\!]^G \cap N_G \times N_G) \cup \{(a,a) \mid a \in N - N_G\}$.*

Proof. By induction. If E is a property name or its inverse, then the claim clearly holds. Now assume E is of the form $E_1 \cup E_2$. The cases where both E_1 and E_2 are safe, or both are unsafe, are clear by induction. If E_1 is safe but E_2 is not, then $[\![E]\!]^G = [\![E_1]\!]^G \cup [\![E_2]\!]^G = ([\![E_1]\!]^G \cap N_G \times N_G) \cup ([\![E_2]\!]^G \cap N_G \times N_G) \cup \{(a,a) \mid a \in N - N_G\} = [\![E]\!]^G \cap (N_G \times N_G) \cup \{(a,a) \mid a \in N - N_G\}$. The same reasoning can be used when E_2 is safe but E_1 is not.

Next, assume E is of the form E_1/E_2. Furthermore assume E_1 is safe, so that E is safe. Let $(x,y) \in [\![E]\!]^G$. Then there exists z such that $(x,z) \in [\![E_1]\!]^G$ and $(z,y) \in [\![E_2]\!]^G$. Since E_1 is safe, x and z are in N_G. Now regardless of whether E_2 is safe or not, since $(z,y) \in [\![E_2]\!]^G$ and $z \in N_G$, we get $y \in N_G$ as desired. The same reasoning can be used when E_2 is safe.

If E is not safe, we verify that $[\![E]\!]^G = ([\![E]\!]^G \cap N_G \times N_G) \cup \{(a,a) \mid a \in N - N_G\}$. For the inclusion from left to right, take $(x,y) \in [\![E]\!]^G$. Then there exists z such that $(x,z) \in [\![E_1]\!]^G$ and $(z,y) \in [\![E_2]\!]^G$. By induction, there are four cases. If both (x,z) and (z,y) are in $N_G \times N_G$, then clearly $(x,y) \in [\![E]\!]^G \cap N_G \times N_G$. If both $(x,z),(z,y)$ are in $\{(a,a) \mid a \in N - N_G\}$ clearly $(x,y) \in \{(a,a) \mid a \in N - N_G\}$. Lastly, the two cases where one of (x,z) and (z,y) is in $N_G \times N_G$ and the other in $\{(a,a) \mid a \in N - N_G\}$, are not possible.

For the inclusion from right to left, take $(x,y) \in [\![E]\!]^G \cap (N_G \times N_G) \cup \{(a,a) \mid a \in N - N_G\}$. If $(x,y) \in [\![E]\!]^G \cap N_G \times N_G$ then $(x,y) \in [\![E]\!]^G$. Otherwise, $(x,y) = (a,a)$ for some $a \in N - N_G$. Then $(a,a) \in [\![E_1]\!]^G$ and $(a,a) \in [\![E_2]\!]^G$ since E_1 and E_2 are not safe. We conclude $(a,a) \in [\![E_1/E_2]\!]^G$ as desired.

Next, assume E is of the form E_1^*. Note that E is unsafe. By definition of Kleene star, we only need to verify that $[\![E]\!]^G \subseteq ([\![E]\!]^G \cap N_G \times N_G) \cup \{(a,a) \mid a \in N - N_G\}$. Let $(x,y) \in [\![E]\!]^G$. If $x = y$, the claim clearly holds. Otherwise, we consider two cases:

- If E_1 is safe, we know $[\![E_1]\!]^G \subseteq N_G \times N_G$. Clearly the reflexive-transitive closure of a subset of $N_G \times N_G$ is also a subset of $N_G \times N_G$. Therefore, $(x,y) \in N_G \times N_G$ as desired.

- If E_1 is unsafe, then by induction $[\![E_1]\!]^G = ([\![E_1]\!]^G \cap N_G \times N_G) \cup \{(a,a) \mid a \in N - N_G\}$. As $x \neq y$ we know (x,y) is in the reflexive-transitive closure of $[\![E_1]\!]^G \cap N_G \times N_G$ which is a subset of $N_G \times N_G$. □

Lastly, we define the notion of a *string*, together with the following Lemma, detailing a convenient property of path expressions.

Definition 4.6. A string s is a path expression of the form: *id*, or s'/p or s'/p^- where s' is a string and p is a property name.

Lemma 4.7. *For every path expression E and every natural number n, there exists a finite non-empty set of strings U s.t. for every graph G with at most n nodes we have $[\![E]\!]^G = \bigcup_{s \in U} [\![s]\!]^G$.*

Before we proceed with the proof, we first state an auxiliary lemma:

Lemma 4.8. *Let V be a finite set of n elements, and let $R \subseteq V \times V$ be a binary relation over V. We have $R^* = R^0 \cup R^1 \cup \cdots \cup R^{n-1}$.*

Proof. R^* is defined as $R^0 \cup R^1 \cup \ldots$ however, we will show that if $(a, b) \in R^m$, with $m \geq n$, then there exists a $k < m$ such that $(a, b) \in R^k$.

We call a sequence of elements $x_1, \ldots, x_h$ an R-path if $(x_l, x_{l+1}) \in R$ for $1 \leq l \leq h$.

If $(a, b) \in R^m$, then there exists an R-path $x_1, \ldots, x_{m+1}$ with $x_1 = a$ and $x_{m+1} = b$. As there are only n total elements, there exists i, j with $1 \leq i < j \leq m + 1$ such that $x_i = x_j$. Therefore, $x_1, \ldots, x_{i-1}, x_j, \ldots, x_{m+1}$ is also an R-path. We conclude that $(a, b) \in R^{m-(j-i)}$, as desired. □

Proof of Lemma 4.7. The proof is by induction on the structure of E. Clearly for the base case $E = p$, we have the set $U = \{p\}$ and similarly for $E = p^-$ we have $U = \{p^-\}$. When $E = id$, clearly $U = \{id\}$. Next, we consider the inductive cases. When $E = E_1 \cup E_2$, we know by induction there exists a set of strings U_1 for E_1, and U_2 for E_2. We then have $U = U_1 \cup U_2$. When $E = E_1/E_2$, we again know by induction there exists a set of strings U_1 for E_1, and U_2 for E_2. We have $U = \{s_1/s_2 \mid s_1 \in U_1 \text{ and } s_2 \in U_2\}$. Finally, when $E = E'^*$, we know by induction there exists a set of strings U' for E'. Let W be a set of strings, we define $W^1 := W$, and for a natural number $m > 1$, $W^m := \{s_1/s_2 \mid s_1 \in W, s_2 \in W^{m-1}\}$. We also use the shorthand notation E^m, with $m > 0$ a natural number, to denote m compositions of the path expression E. For example, E^3 is $E/E/E$. By definition, $[\![E'^*]\!]^G = [\![id]\!]^G \cup [\![E']\!]^G \cup [\![E'^2]\!]^G \cup \ldots$. By Lemma 4.8 we know that this is the same as $[\![E'^*]\!]^G = [\![id]\!]^G \cup [\![E']\!]^G \cup [\![E'^2]\!]^G \cup \cdots \cup [\![E'^{n-1}]\!]^G$ for graphs with at most n nodes. It then follows that $U = \{id\} \cup U' \cup U'^2 \cup \cdots \cup U'^{n-1}$. □

4.1.2 Disjointness

We present here the proof for $X = disj$. The general strategy is to first characterize the behavior of path expressions on G and G'. Then the Proposition is proven with a stronger induction hypothesis, to allow the induction to carry through. A similar strategy is followed in the proof for $X = eq$.

We begin by defining the graphs G and G' more formally.

Definition 4.9 ($G_{disj}(\Sigma, m)$)**.** Let Σ be a finite vocabulary including r, and let m be a natural number. We define the graph $G_{disj}(\Sigma, m)$ over the set of property names in Σ as follows. Let $M = \max(m, 3)$. There are $4M$ nodes in the graph, which are chosen outside of Σ. We denote these nodes by x_i^j for $i = 1, 2, 3, 4$ and $j = 1, \ldots, M$. (In the description that follows, subscripts range from 1 to 4 and superscripts range from 1 to M.) For each property name p in Σ, the graph has the same set of p-edges. We describe these edges next. There is an edge from x_i^j to $x_{i \bmod 4+1}^{j'}$ for every i, j and j'. Moreover, if i is 2 or 4, there is an edge from x_i^j to $x_i^{j'}$ for all $j \neq j'$. So, formally, we have: $G_{disj}(\Sigma, m) := \{(x_i^j, p, x_{i \bmod 4+1}^{j'}) \mid i \in \{1, \ldots, 4\} \text{ and } j, j' \in \{1, \ldots, M\} \text{ and } p \in \Sigma \cap P\} \cup \{(x_i^j, p, x_i^{j'}) \mid i \in \{1, \ldots, 4\} \text{ and } j, j' \in \{1, \ldots, M\} \text{ and } j \neq j' \text{ and } p \in \Sigma \cap P\}$.

Thus, in Figure 4.1, bottom left, one can think of the left oval as the set of nodes x_1^j; the top cloud as the set of nodes x_2^j; and so on. We call the nodes x_i^j with $i = 2, 4$ the *even nodes*, and the nodes x_i^j with $i = 1, 3$ the *odd nodes*.

Definition 4.10 ($G'_{disj}(\Sigma, m)$)**.** We define the graph $G'_{disj}(\Sigma, m)$ in the same way as $G_{disj}(\Sigma, m)$ except that there is an edge from x_i^j to $x_i^{j'}$ for all i and $j \neq j'$ (not only for even i values).

We characterize the behavior of path expressions on the graph $G_{disj}(\Sigma, m)$ as follows.

Lemma 4.11. *Let G be $G_{disj}(\Sigma, m)$. Call a path expression* simple *if it is a union of expressions of the form $s_1/\ldots/s_n$, where $n \geq 1$ and one of the s_i is a property name while the other s_i are "id". Let E be a non-simple, id-free path expression over Σ. The following three statements hold:*

1. *(A) for all even nodes v of G, we have $[\![E]\!]^G(v) \supseteq [\![r]\!]^G(v)$; or*

   (B) for all even nodes v of G, we have $[\![E]\!]^G(v) \supseteq [\![r^-]\!]^G(v)$.

2. *(C) for all odd nodes v of G, we have $[\![E]\!]^G(v) \supseteq [\![r]\!]^G(v)$; or*

   (D) for all odd nodes v of G, we have $[\![E]\!]^G(v) \supseteq [\![r^-]\!]^G(v)$.

3. *For all nodes v of G, we have $[\![E]\!]^G(v) - [\![r]\!]^G(v) \neq \emptyset$.*

Proof. For $i = 1, 2, 3, 4$, define the *i-th blob of nodes* to be the set $X_i = \{x_i^1, \ldots, x_i^M\}$ (see Figure 4.1). We also use the notations *next*$(1) = 2$; *next*$(2) = 3$; *next*$(3) = 4$; *next*$(4) = 1$; *prev*$(4) = 3$; *prev*$(3) = 2$; *prev*$(2) = 1$; *prev*$(1) = 4$. Thus *next*(i) indicates the next blob in the cycle, and *prev*(i) the previous.

The proof is by induction on the structure of E. If E is a property name, E is simple so the claim is trivial. If E is of the form p^-, cases B and D are clear and we only need to verify the third statement. That holds because for any i, if $v \in X_i$, then $[\![p^-]\!]^G(v) \supseteq X_{prev(i)}$ and clearly $X_{prev(i)} - [\![r]\!]^G(v) \neq \emptyset$. We next consider the inductive cases.

First, assume E is of the form $E_1 \cup E_2$. When at least one of E_1 and E_2 is not simple, the three statements immediately follow by induction, since $[\![E]\!]^G \supseteq [\![E_1]\!]^G$ and $[\![E]\!]^G \supseteq [\![E_2]\!]^G$. If E_1 and E_2 are simple, then E is simple and the claim is trivial.

Next, assume E is of the form E_1^*. If E_1 is not simple, the three statements follow immediately by induction, since $[\![E]\!]^G \supseteq [\![E_1]\!]^G$. If E_1 is simple, cases A and C clearly hold for E, so we only need to verify the third statement. That holds because, by the form of E, every node v is in $[\![E]\!]^G(v)$, but not in $[\![r]\!]^G(v)$, as G does not have any self-loops.

Finally, assume E is of the form E_1/E_2. Note that if E_1 or E_2 is simple, clearly cases A and C apply to them. The argument that follows will therefore also apply when E_1 or E_2 is simple. We will be careful not to apply the induction hypothesis for the third statement to E_1 and E_2.

We first focus on the even nodes, and show the first and the third statement. We distinguish two cases.

- If case A applies to E_2, then we show that case A also applies to E. Let $v \in X_i$ be an even node. We verify the following two inclusions:
  - $[\![E]\!]^G(v) \supseteq X_i$. Let $u \in X_i$. If $u \neq v$, choose a third node $w \in X_i$. Since X_i is a clique, $(v, w) \in [\![E_1]\!]^G$ regardless of whether case A or B applies to E_1. By case A for E_2, we also have $(w, u) \in [\![E_2]\!]^G$, whence $u \in [\![E]\!]^G(v)$ as desired. If $u = v$, we similarly have $(v, w) \in [\![E_1]\!]^G$ and $(w, u) \in [\![E_2]\!]^G$ as desired.
  - $[\![E]\!]^G(v) \supseteq X_{next(i)}$. Let $u \in X_{next(i)}$ and choose $w \neq v \in X_i$. Regardless of whether case A or B applies to E_1, we have $(v, w) \in [\![E_1]\!]^G$. By case A for E_2, we also have $(w, u) \in [\![E_2]\!]^G$, whence $u \in [\![E]\!]^G(v)$ as desired.

We conclude that $[\![E]\!]^G(v) \supseteq X_i \cup X_{next(i)} \supseteq [\![r]\!]^G$ as desired.

- If case B applies to E_2, then we show that case B also applies to E. This is analogous to the previous case, now verifying that $[\![E]\!]^G(v) \supseteq X_i \cup X_{prev(i)}$.

In both cases, the third statement now follows for even nodes v. Indeed, $v \in X_i \subseteq [\![E]\!]^G(v)$ but $v \notin [\![r]\!]^G(v)$.

We next focus on the odd nodes, and show the second and the third statement. We again consider two cases.

- If case C applies to E_1, then we show that case C also applies to E. Let $v \in X_i$ be an odd node. Note that $[\![r]\!]^G(v) = X_{next(i)}$. To verify that $[\![E]\!]^G(v) \supseteq X_{next(i)}$, let $u \in X_{next(i)}$. Then u is even. Choose $w \neq u \in X_{next(i)}$. Since case C applies to E_1, we have $(v, w) \in [\![E_1]\!]^G$. Moreover, since $X_{next(i)}$ is a clique, $(w, u) \in [\![E_2]\!]^G$ regardless of whether case A or B applies to E_2. We obtain $(v, u) \in [\![E]\!]^G$ as desired.

  We also verify the third statement for odd nodes in this case. We distinguish two further cases.

  - If case A applies to E_2, any node $u \in X_{next(next(i))}$ belongs to $[\![E]\!]^G(v)$, and clearly these u are not in $X_{next(i)} = [\![r]\!]^G(v)$.
  - If case B applies to E_2, then, since X_i is a clique, any node $u \in X_i$ belongs to $[\![E]\!]^G(v)$, and again these u are not in $X_{next(i)}$.

- If case D applies to E_1, then we show that case D also applies to E. This is analogous to the previous case, now verifying that $[\![E]\!]^G(v) \supseteq X_{prev(i)}$. In this case the third statement for odd nodes is clear, as clearly $X_{prev(i)} - X_{next(i)} \neq \emptyset$. □

We similarly characterize the behavior of path expressions on the other graph.

Lemma 4.12. *Let G' be $G'_{disj}(\Sigma, m)$ and let E be a non-simple, id-free path expression over Σ. The following statements hold:*

1. $[\![E]\!]^{G'} \supseteq [\![r]\!]^{G'}$ *or* $[\![E]\!]^{G'} \supseteq [\![r^-]\!]^{G'}$.
2. *For all nodes v of G', we have* $[\![E]\!]^{G'}(v) - [\![r]\!]^{G'}(v) \neq \emptyset$.

Proof. For $i = 1, 2, 3, 4$, define the *i-th blob of nodes* to be the set $X_i = \{x_i^1, \ldots, x_i^M\}$ (see Figure 4.1). We also use the notations $next(1) = 2$; $next(2) = 3$; $next(3) = 4$; $next(4) = 1$; $prev(4) = 3$; $prev(3) = 2$; $prev(2) = 1$; $prev(1) = 4$. Thus $next(i)$ indicates the next blob in the cycle, and $prev(i)$ the previous.

The proof is by induction on the structure of E. If E is a property name, E is simple so the claim is trivial. If E is of the form p^-, the first claim is clear because $[\![r^-]\!]^{G'} \subseteq [\![E]\!]^{G'}$, and we only need to verify the second one. That holds because for any i, if $v \in X_i$, then $[\![p^-]\!]^{G'}(v) \supseteq X_{prev(i)}$ and clearly $X_{prev(i)} - [\![r]\!]^{G'}(v) \neq \emptyset$. We next consider the inductive cases.

First, assume E is of the form $E_1 \cup E_2$. When at least one of E_1 and E_2 is not simple, the two claims immediately follow by induction, since $[\![E]\!]^{G'} \supseteq [\![E_1]\!]^{G'}$ and $[\![E]\!]^{G'} \supseteq [\![E_2]\!]^{G'}$. If E_1 and E_2 are simple, then E is simple and the claim is trivial.

Next, assume E is of the form E_1^*. If E_1 is not simple, the two claims follow immediately by induction, since $[\![E]\!]^{G'} \supseteq [\![E_1]\!]^{G'}$. If E_1 is simple, the first claim clearly hold for E, so we only need to verify the second claim. That holds because, by the form of E, every node v is in $[\![E]\!]^{G'}(v)$, but not in $[\![r]\!]^{G'}(v)$, as G does not have any self-loops.

Finally, assume E is of the form E_1/E_2. Note that if E_1 or E_2 is simple, clearly claim one holds because $[\![r]\!]^{G'} \subseteq [\![E]\!]^{G'}$. The argument that follows will therefore also apply when E_1 or E_2 is simple. We will be careful not to apply the induction hypothesis for the second statement to E_1 and E_2.

We distinguish two cases.

- If $[\![r]\!]^{G'} \subseteq [\![E_2]\!]^{G'}$, then we show that $[\![r]\!]^{G'} \subseteq [\![E]\!]^{G'}$. Let $v \in X_i$. We verify the following two inclusions:
  - $[\![E]\!]^{G}(v) \supseteq X_i$. Let $u \in X_i$. If $u \neq v$, choose a third node $w \in X_i$. Since X_i is a clique, $(v, w) \in [\![E_1]\!]^G$ because the first claim holds for E_1. By $[\![r]\!]^{G'} \subseteq [\![E_2]\!]^{G'}$, we also have $(w, u) \in [\![E_2]\!]^{G'}$, whence $u \in [\![E]\!]^{G'}(v)$ as desired. If $u = v$, we similarly have $(v, w) \in [\![E_1]\!]^{G'}$ and $(w, u) \in [\![E_2]\!]^{G'}$ as desired.
  - $[\![E]\!]^{G}(v) \supseteq X_{next(i)}$. Let $u \in X_{next(i)}$ and choose $w \neq v \in X_i$. Because the first claim holds for E_1, we have $(v, w) \in [\![E_1]\!]^G$. By $[\![r]\!]^{G'} \subseteq [\![E_2]\!]^{G'}$, we also have $(w, u) \in [\![E_2]\!]^{G'}$, whence $u \in [\![E]\!]^{G'}(v)$ as desired.

  We conclude that $[\![E]\!]^{G'}(v) \supseteq X_i \cup X_{next(i)} \supseteq [\![r]\!]^{G'}$ as desired.

- If $[\![r^-]\!]^{G'} \subseteq [\![E_2]\!]^{G'}$, then we show that $[\![r^-]\!]^{G'} \subseteq [\![E]\!]^{G'}$. This is analogous to the previous case, now verifying that $[\![E]\!]^{G}(v) \supseteq X_i \cup X_{prev(i)}$.

In both cases, the second statement now follows for every node v. Indeed, $v \in X_i \subseteq [\![E]\!]^{G'}(v)$ but $v \notin [\![r]\!]^{G'}(v)$. □

We are now ready to prove the non-obvious part of Proposition 4.2 where $X = disj$. We use the following version of the proposition.

Proposition 4.13. *Let V be the common set of nodes of the graphs $G = G_{disj}(\Sigma, m)$ and $G' = G'_{disj}(\Sigma, m)$. Let φ be a shape over Σ that does not use disj, and that counts to at most m. Then either $[\![\varphi]\!]^G \cap V = \emptyset$ or $[\![\varphi]\!]^G \supseteq V$. Moreover, $[\![\varphi]\!]^G = [\![\varphi]\!]^{G'}$.*

Proof. This is proven by induction on the structure of φ. Let H be G or G'. If φ is $\top$, then $[\![\top]\!]^H = N \supseteq V$. If φ is $hasValue(c)$, then $[\![hasValue(c)]\!]^H = \{c\} \subseteq \Sigma$ and we know that $\Sigma \cap V = \emptyset$. Next assume φ is of the form $eq(E, p)$. Using Lemma 4.3, we distinguish four different cases for E.

- E is *id*. According to Lemma 4.11 and Lemma 4.12 $[\![E]\!]^H$ will always contain either $[\![p]\!]^H$ or $[\![p^-]\!]^H$. In both cases, $[\![E]\!]^H(v)$ clearly never equals $[\![id]\!]^H(v) = \{v\}$. Therefore, $[\![\varphi]\!]^H \cap V = \emptyset$.

- E is $E' \cup id$ where E' is *id*-free or E itself is *id*-free and non-simple. Lemmas 4.11 and 4.12 tell us that $[\![E]\!]^H(v) - [\![r]\!]^H(v) \neq \emptyset$ for every $v \in V$. Since $[\![r]\!]^H = [\![p]\!]^H$, this means $H, v \not\models \varphi$ for $v \in V$, or, equivalently, $[\![\varphi]\!]^G \cap V = \emptyset$. To see that, moreover, $[\![\varphi]\!]^G = [\![\varphi]\!]^{G'}$, it remains to show that $G, v \models \varphi$ iff $G', v \models \varphi$ for all node names $v \notin V$.

- E is *id*-free and simple. Then $[\![E]\!]^H = [\![p]\!]^H$, so clearly $[\![\varphi]\!]^H = N \supseteq V$.

We still need to show $[\![\varphi]\!]^G = [\![\varphi]\!]^{G'}$. Clearly, $[\![p]\!]^G(v) = [\![p]\!]^{G'}(v) = \emptyset$. Now by Lemma 4.5, if E is safe, then also $[\![E]\!]^G(v) = [\![E]\!]^{G'}(v) = \emptyset$, so $G, v \models \varphi$ and $G', v \models \varphi$. On the other hand, if E is unsafe, then by the same Lemma $[\![E]\!]^G(v) = [\![E]\!]^{G'}(v) = \{v\} \neq \emptyset$, so $G, v \not\models \varphi$ and $G', v \not\models \varphi$, as desired.

As the final base case, assume φ is of the form $closed(R)$. If Σ contains a property name p not in R, then $[\![\varphi]\!]^H \cap V = \emptyset$, since every node in H has an outgoing p-edge. Otherwise, i.e., if $\Sigma \subseteq R$, we have $[\![\varphi]\!]^H \supseteq V$, since every node in H has only outgoing edges labeled by property names in Σ. To see that, moreover, $[\![\varphi]\!]^G = [\![\varphi]\!]^{G'}$, it suffices to observe that trivially $H, v \models \varphi$ for all node names $v \notin V$.

We next consider the inductive cases. The cases for the boolean connectives follow readily by induction. Finally, assume φ is of the form $\geq_n E.\varphi_1$. By induction, there are two possibilities for φ_1:

- If $[\![\varphi_1]\!]^H \cap V = \emptyset$, then also $[\![\varphi]\!]^H \cap V = \emptyset$ since path expressions can only reach nodes in some graph from nodes in that graph.

- If $[\![\varphi_1]\!]^H \supseteq V$, we distinguish three cases using Lemma 4.3. First, when E is *id*, then if $n = 1$, $[\![\varphi]\!]^H \supseteq V$. Otherwise, if $n \neq 1$, then $[\![\varphi]\!]^H = \emptyset$. Next, when E is *id*-free or $E' \cup id$ with E' an *id*-free path expression, it suffices to show that $\sharp[\![E']\!]^H(v) \geq n$ for all $v \in V$. By Lemmas 4.11 and 4.12 we know that $[\![E_1]\!]^H(v)$ contains $[\![r]\!]^H(v)$ or $[\![r^-]\!]^H(v)$. Inspecting H, we see that each of these sets has at least $\max(3, m) \geq n$ elements, as desired. Finally, when E is equivalent to an *id*-free path expression or whenever E simply does not use *id*, the argument is analogous to the previous case.

In both cases we still need to show that $[\![\varphi]\!]^G = [\![\varphi]\!]^{G'}$. We already showed that $[\![\varphi]\!]^G \supseteq V$ and $[\![\varphi]\!]^{G'} \supseteq V$, or $[\![\varphi]\!]^G \cap V = \emptyset$ and $[\![\varphi]\!]^{G'} \cap V = \emptyset$. Therefore, towards a proof of the equality, we only need to consider the node names not in V.

For the inclusion from left to right, take $x \in [\![\varphi]\!]^G - V$. Since $G, x \models \varphi$, there exist y_1, $\dots$, y_n such that $(x, y_i) \in [\![E]\!]^G$ and $G, y_i \models \varphi_1$ for $i = 1, \dots, n$. However, since $x \notin V$, by Lemma 4.5, all y_i must equal x. Hence, $n = 1$ and $(x, x) \in [\![E]\!]^G$ and $G, x \models \varphi_1$. Then again by the same Lemma, $(x, x) \in [\![E]\!]^{G'}$, since G and G' have the same set of nodes V. Moreover, by induction, $G', x \models \varphi_1$. We conclude that $G', x \models \varphi$ as desired. The inclusion from right to left is argued symmetrically. □

4.1.3 Equality

Next, we turn our attention to Proposition 4.2 for $X = eq$. We define the graphs from Figure 4.1 formally.

Definition 4.14. Let Σ be a finite vocabulary including r, and let m be a natural number. Choose a set V of node names outside Σ, of cardinality $M := \max(3, m + 1)$. Fix two arbitrary nodes a and b from V. We define the graph $G_{eq}(\Sigma)$ over the set of property names from Σ as follows. For each property name p in Σ, the set of p-edges in $G_{eq}(\Sigma)$ equals $V \times V - (b, a)$. We define the graph $G'_{eq}(\Sigma)$ similarly, but with $V \times V$ as the set of p-edges.

So, $G'_{eq}(\Sigma, m)$ is a complete graph, and $G_{eq}(\Sigma, m)$ is a complete graph with one edge (b, a) removed.

Lemma 4.15. *Let E be an id-free path expression over Σ and let $H = G_{eq}(\Sigma, m)$ or $G'_{eq}(\Sigma, m)$. Then*

A. $[\![E]\!]^H \supseteq [\![r]\!]^H$*, or*

B. $[\![E]\!]^H \supseteq [\![r^-]\!]^H$.

Proof. The claim is obvious for $G'_{eq}(\Sigma, m)$, being a complete graph. So we focus on the graph $G_{eq}(\Sigma, m)$. The proof is by induction. If E is a property name or its inverse, the claim is clear. If E is of the form $E_1 \cup E_2$, the claim is immediate by induction.

Assume E is of the form E_1/E_2. We show that A applies.[1] If A applies to E_1, this is clear, since we can follow any edge by E_1 and then stay at the head of the edge by E_2 using

[1] Actually, $[\![E]\!]^G$ always contains $V \times V$ in this case, but we do not need this.

the self-loop. If B applies to E_1, the same can still be done for all edges except for (a, b), which is the only nonsymmetrical edge. To go from a to b by E, we go by E_1 from a to a node c distinct from a and b, then go by E_2 from c to b.

If E is of the form E_1^*, again A applies, since E_1^* contains E_1/E_1. □

We are now ready to prove the non-obvious part of Proposition 4.2 where $X = \mathit{eq}$. We use the following version of the proposition.

Proposition 4.16. *Let G be $G_{eq}(\Sigma, m)$ and let G' be $G'_{eq}(\Sigma, m)$. Let φ be a shape over Σ that does not use eq and that counts to at most m. Then either $[\![\varphi]\!]^G \cap V = \emptyset$ or $[\![\varphi]\!]^G \supseteq V$. Moreover, $[\![\varphi]\!]^G = [\![\varphi]\!]^{G'}$.*

Proof. This is proven by induction on the structure of φ. Let H be G or G'. We focus directly on the relevant cases. Assume φ is of the form $\mathit{disj}(E_1, E_2)$. Lemma 4.15 clearly yields that $[\![\varphi]\!]^H \cap V = \emptyset$. It again remains to verify that $G, v \models \varphi$ iff $G', v \models \varphi$ for all node names $v \notin V$. By Lemma 4.5, for such v and $H = G$ or G', we indeed have $H, v \models \varphi$ if exactly one of E_1 and E_2 is safe. If both are safe or both are unsafe, we have $H, v \not\models \varphi$.

The last base case of interest is the case where φ is of the form $\mathit{closed}(R)$. This goes again exactly as in the proof for $X = \mathit{disj}$.

We next consider the inductive cases. The cases for the boolean connectives follow readily by induction. Finally, assume φ is of the form $\geq_n E.\varphi_1$. By induction, there are two possibilities for φ_1:

- If $[\![\varphi_1]\!]^G \cap V = \emptyset$ then $[\![\varphi]\!]^G \cap V = \emptyset$ since path expressions can only reach nodes in some graph from nodes in that graph.

- If $[\![\varphi_1]\!]^H \supseteq V$, we distinguish three cases using Lemma 4.3. First, when E is id, then if $n = 1$, $[\![\varphi]\!]^H \supseteq V$. Otherwise, if $n \neq 1$, then $[\![\varphi]\!]^H = \emptyset$. Next, when E is id-free or $E' \cup \mathit{id}$ with E' an id-free path expression, it suffices to show that $\sharp[\![E']\!]^H(v) \geq n$ for all $v \in V$. By Lemma 4.15, we know that $[\![E]\!]^H(v)$ contains $[\![r]\!]^H(v)$ or $[\![r^-]\!]^H(v)$. These sets contain at least $M - 1 \geq m \geq n$ elements as desired. (The number $M - 1$ is reached only when H is G and $v = b$ or $v = a$; otherwise the sets contain M elements.)

The equality $[\![\varphi]\!]^G = [\![\varphi]\!]^{G'}$ is shown in the same way as in the proof for $X = \mathit{disj}$ (Section 4.1.2). □

4.1.4 Closure

Without using *closed*, shapes cannot say anything about properties that they do not explicitly mention. We formalize this intuitive observation as follows. The proof is straightforward.

Lemma 4.17. *Let Σ be a vocabulary, let E be a path expression over Σ, and let φ be a shape over Σ that does not use closed. Let G_1 and G_2 be graphs such that $[\![p]\!]^{G_1} = [\![p]\!]^{G_2}$ for every property name p in Σ. Then $[\![E]\!]^{G_1} = [\![E]\!]^{G_2}$ and $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2}$.* □

Theorem 4.1 now follows readily for $X = \mathit{closed}$. Let F be a feature set without *closed*, let $\mathcal{S}$ be a shape schema in $\mathcal{L}(F)$, and let φ be the validation shape of $\mathcal{S}$. Let p be a property name not mentioned in $\mathcal{S}$, and different from r. Consider the graphs $G = \{(a, r, a), (a, p, a)\}$ and $G' = \{(a, r, a)\}$, so that G' belongs to Q_{closed} but G does not. By Lemma 4.17 we have $[\![\varphi]\!]^G = [\![\varphi]\!]^{G'}$, showing that $\mathcal{S}$ does not define Q_{closed}.

Remark 4.18. Lemma 4.17 fails completely in the presence of closure constraints. The simplest counterexample is to consider $\Sigma = \emptyset$ and the shape $\mathit{closed}(\emptyset)$. Trivially, any two graphs agree on the property names from Σ. However, $[\![\mathit{closed}(\emptyset)]\!]^G$, which equals the set of node names that do not have an outgoing edge in G (they may still have an incoming edge), obviously depends on the graph G.

The reader may wonder if this statement still holds under active domain semantics. In such semantics, which we denote by $[\![\varphi]\!]^G_{adom}$, we would view G as an interpretation with domain *not* the whole of N; rather we would take as domain the set $N_G \cup C$, with C the set of constants mentioned in φ. When assuming active domain semantics, a modified lemma is required. To see this, consider the graph $G = \{(a,p,b)\}$ and $G' = \{(a,p,b),(a,q,c)\}$. Let φ simply be $\top$. We have $[\![\varphi]\!]^G_{adom} = \{a,b\}$ and $[\![\varphi]\!]^{G'}_{adom} = \{a,b,c\}$, so Lemma 4.17 no longer holds. We can, however, give the following more refined variant of Lemma 4.17:

Lemma 4.19. *Let Σ be a vocabulary, let E be a path expression over Σ, and let φ be a shape over Σ that does not use closed. Let I_1 and I_2 be interpretations such that $[\![p]\!]^{I_1} = [\![p]\!]^{I_2}$ for every property name p in Σ. Then $[\![E]\!]^{I_1} \cap \Delta^{I_2} \times \Delta^{I_2} = [\![E]\!]^{I_2} \cap \Delta^{I_1} \times \Delta^{I_1}$ and $[\![\varphi]\!]^{I_1} \cap \Delta^{I_2} = [\![\varphi]\!]^{I_2} \cap \Delta^{I_1}$.* □

The same reasoning as given after Lemma 4.17, now using the new Lemma, then shows that *closed* is still primitive under active domain semantics.

4.2 Are target-based shape schemas enough?

Lemma 4.17 also allows us to clarify that, as far as expressive power is concerned, and in the absence of closure constraints, the restriction to target-based shape schemas is inconsequential.

Theorem 4.20. *Every generalized shape schema that does not use closure constraints is equivalent to a target-based shape schema (that still does not use closure constraints).*

In order to prove this theorem, we first establish the following lemma.

Lemma 4.21. *Let φ be a shape and let C be the set of constants mentioned in φ. Assume there exists a graph G and a node name $x \notin N_G \cup C$ such that $G, x \models \varphi$. Then for any graph H and any node name $y \notin N_H \cup C$, also $H, y \models \varphi$.*

Proof. By induction on φ. The case where φ is of the form *hasValue*(c) cannot occur, and the case where φ is $\top$ is trivial.

If φ is $\varphi_1 \vee \varphi_2$ or $\neg\varphi_1$, the claim follows readily by induction.

Now assume φ is of the form $\geq_n E.\varphi_1$. Then there exists $x_1, \dots, x_n$ such that $(x, x_i) \in [\![E]\!]^G$ and $G, x_i \models \varphi_1$ for $i = 1, \dots, n$. However, since $x \notin N_G$, by Lemma 4.5, all x_i must equal x. Hence, $n = 1$ and $(x,x) \in [\![E]\!]^G$ and $G, x \models \varphi_1$. By the same Lemma, $(y,y) \in [\![E]\!]^H$, since $y \notin N_H$. Furthermore, by induction, $H, y \models \varphi_1$. We conclude that $H, y \models \varphi$ as desired.

Next, assume φ is *eq*(E,p). Since $G, x \models \varphi$, but $[\![p]\!]^G = \emptyset$ since $x \notin N_G$, also $[\![E]\!]^G(x) = \emptyset$. Then by Lemma 4.5, also $[\![E]\!]^H(y) = \emptyset$, since $y \notin N_H$. Furthermore, also $[\![p]\!]^H(y) = \emptyset$. We conclude that $H, y \models \varphi$ as desired.

Next assume φ is *disj*(E,p). Then $H, y \models \varphi$ is clear. Indeed, since $y \notin N_H$, we have $[\![p]\!]^H(y) = \emptyset$.

Finally, assume φ is *closed*(R). Then again $H, y \models \varphi$ is clear because $y \notin N_H$. □

We can now show the theorem.

Proof of Theorem 4.20. Let φ be the validation shape for shape schema $\mathcal{S}$, so that $G \models \mathcal{S}$ if and only if $[\![\varphi]\!]^G$ is empty. Let C be the set of constants mentioned in φ.

Let us say that φ is *internal* if for every graph G and every node name v such that $G, v \models \varphi$, we have $v \in N_G \cup C$. If φ is not internal, then, using Lemma 4.21, for every graph G and every node $v \notin N_G \cup C$, we have $G, v \models \varphi$. Thus, if φ is not internal, $\mathcal{S}$ is unsatisfiable and is equivalent to the single target-based inclusion *hasValue*$(c) \subseteq \neg\top$, for an arbitrary constant c.

So now assume φ is internal. Define the target-based shape schema $\mathcal{T}$ consisting of the following inclusions:

- For each constant $c \in C$, the inclusion $\mathit{hasValue}(c) \subseteq \neg\varphi$.
- For each property name mentioned in φ, the two inclusions $\exists p.\top \subseteq \neg\varphi$ and $\exists p^-.\top \subseteq \neg\varphi$.

We will show that $\mathcal{S}$ and $\mathcal{T}$ are equivalent. Let ψ be the validation shape for $\mathcal{T}$.

Let G be any graph, and let G' be the graph obtained from G by removing all triples involving property names not mentioned in φ. We reason as follows:

$$\begin{aligned} G \models \mathcal{S} &\Leftrightarrow [\![\varphi]\!]^G = \emptyset \\ &\Leftrightarrow [\![\varphi]\!]^{G'} = \emptyset && \text{by Lemma 4.17} \\ &\Leftrightarrow G' \models \mathcal{T} && \text{since } \varphi \text{ is internal} \\ &\Leftrightarrow [\![\psi]\!]^{G'} = \emptyset \\ &\Leftrightarrow [\![\psi]\!]^{G} = \emptyset && \text{by Lemma 4.17} \\ &\Leftrightarrow G \models \mathcal{T} \end{aligned}$$

□

Remark 4.22. Note that we do not need class-based targets in the proof, so such targets are redundant on the left-hand sides of inclusions. This can also be seen directly: any inclusion

$$\exists \mathsf{type}/\mathsf{subclass}^*.\mathit{hasValue}(c) \subseteq \varphi$$

with a class-based target is equivalent to the following inclusion with a subjects-of target:

$$\exists \mathsf{type}.\top \subseteq \neg\exists \mathsf{type}/\mathsf{subclass}^*.\mathit{hasValue}(c) \vee \varphi$$

Remark 4.23. Theorem 4.20 fails in the presence of closure constraints. For example, the inclusion $\neg \mathit{closed}(\emptyset) \subseteq \exists r.\top$ defines the class of graphs where every node with an outgoing edge has an outgoing r-edge. Suppose this inclusion would be equivalent to a target-based shape schema $\mathcal{S}$, and let R be the set of all property names mentioned in the targets of $\mathcal{S}$. Let p be a property name not in R and distinct from r; let a be a node name not used as a constant in $\mathcal{S}$; and consider the graph $G = \{(a, p, a)\}$. This graph trivially satisfies $\mathcal{S}$, but violates the inclusion.

4.3 Extensions for full equality and disjointness tests

A quirk in the design of SHACL is that it only allows equality and disjointness tests $\mathit{eq}(E_1, E_2)$ and $\mathit{disj}(E_1, E_2)$ where E_1 can be a general path expression, but E_2 needs to be a property name. The next question we can ask is whether allowing "full" equality or disjointness tests, i.e., allowing a general path expression for E_2, strictly increases the expressive power. Within the community there are indeed plans to extend SHACL in this direction [56].

When we allow for such "full" equality and disjointness tests, it gives rise to two new features: *full-eq* and *full-disj*. Formally, we extend the grammar of shapes with two new constructs: $\mathit{eq}(E_1, E_2)$ and $\mathit{disj}(E_1, E_2)$.

Remark 4.24. We cannot explicitly write the shapes $\mathit{eq}(\mathit{id}, \mathit{id})$ and $\mathit{disj}(\mathit{id}, \mathit{id})$. However, these shapes are equivalent to $\top$ and $\neg\top$ respectively.

We are going to show that each of these new features strictly adds expressive power. Concretely, we introduce the following classes of graphs.

Full equality $Q_{\mathit{full\text{-}eq}}$ is the class of graphs where all objects of a property name p do not have the same subjects for p and q. Note that $Q_{\mathit{full\text{-}eq}}$ is definable in $\mathcal{L}(\mathit{full\text{-}eq})$ by the single, target-based, inclusion statement $\exists p^-.\top \subseteq \neg \mathit{eq}(p^-, q^-)$.

Listing 4.2: The two shapes used to express the classes of graphs $Q_{\textit{full-eq}}$ and $Q_{\textit{full-disj}}$ in an extension of real SHACL

```
:fullequality a sh:NodeShape ;
  sh:targetSubjectsOf :p ;
  sh:not [  sh:path [ sh:inversePath :p ] ;
            sh:equals [ sh:inversePath :q ] ] .

:fulldisjointness a sh:NodeShape ;
  sh:targetSubjectsOf :p ;
  sh:not [  sh:path [ sh:inversePath :p ] ;
            sh:disjoint [ sh:inversePath :q ] ] .
```

Full disjointness $Q_{\textit{full-disj}}$ is the class of graphs where all objects of a property name p do not have disjoint sets of subjects for p and q. Note that $Q_{\textit{full-disj}}$ is definable in $\mathcal{L}(\textit{full-disj})$ by the single, target-based, inclusion statement $\exists p^-.\top \subseteq \neg \textit{disj}(p^-, q^-)$.

To show the connection to real SHACL, the constraints described above can be written in real SHACL as shown in Listing 4.2.

In the spirit of Theorem 4.1, we are now going to show the following:

Theorem 4.25. *$Q_{\textit{full-eq}}$ is not definable in $\mathcal{L}(\textit{eq}, \textit{full-disj}, \textit{closed})$ and $Q_{\textit{full-disj}}$ is not definable in $\mathcal{L}(\textit{disj}, \textit{full-eq}, \textit{closed})$.*

These two non-definability results are proven in the following Sections 4.3.1 and 4.3.2. Then in Section 4.3.3 we will reconsider the non-definability results for non-full equality and disjointness from Theorem 4.1 in the new light of their full versions.

4.3.1 Full equality

We present here the proof for the primitivity of full equality tests. The general strategy is the same as in Section 4.1, where again we will prove appropriate versions of Proposition 4.2.

We begin by defining the graphs G and G' formally. Note that, as desired, G' belongs to $Q_{\textit{full-eq}}$ but G does not.

Definition 4.26. $G_{\textit{full-eq}}(\Sigma, m)$ Let Σ be a finite vocabulary and let $m \geq 3$ be a natural number. Let $A = \{a_1, \ldots, a_m\}$, $B = \{b_1, \ldots, b_m\}$ and $C = \{c_1, \ldots, c_m\}$ be three disjoint sets of nodes, disjoint from Σ. We define the graph $G_{\textit{full-eq}}(\Sigma, m)$ to be $[\![p]\!]^G = C \times (A \cup B)$ and $[\![q]\!]^G = C \times A \cup \{(c_i, b_j) \mid i \neq j \in \{1, \ldots, m\}\}$.

Definition 4.27. $G'_{\textit{full-eq}}(\Sigma, m)$ We define the graph $G'_{\textit{full-eq}}(\Sigma, m)$ like $G_{\textit{full-eq}}(\Sigma, m)$ but $[\![q]\!]^G = \{(c_i, a_j) \mid i \neq j \in \{1, \ldots, m\}\} \cup \{(c_i, b_j) \mid i \neq j \in \{1, \ldots, m\}\}$.

We identify the possible types of strings on the graphs $G_{\textit{full-eq}}(\Sigma, m)$ and $G'_{\textit{full-eq}}(\Sigma, m)$ as follows.

Lemma 4.28. *Let Σ be a vocabulary. Let $m \geq 3$ be a natural number. Let G be $G_{\textit{full-eq}}(\Sigma, m)$ and let G' be $G'_{\textit{full-eq}}(\Sigma, m)$. The only possibilities for a string s evaluated on G and G' are the following:*

1. $[\![s]\!]^G = [\![p]\!]^G = [\![s]\!]^{G'} = C \times (A \cup B)$.

2. $[\![s]\!]^G = [\![q]\!]^G = (C \times A) \cup \{(c_i, b_j) \mid i \neq j \in \{1, \ldots, m\})\}$ *and*
   $[\![s]\!]^{G'} = [\![q]\!]^{G'} = \{(c_i, a_j) \mid i \neq j \in \{1, \ldots, m\}\} \cup \{(c_i, b_j) \mid i \neq j \in \{1, \ldots, m\}\}$.

3. $[\![s]\!]^G = [\![p^-]\!]^G = [\![s]\!]^{G'} = (A \cup B) \times C$.

4. $[\![s]\!]^G = [\![q^-]\!]^G = (A \times C) \cup \{(b_i, c_j) \mid i \neq j \in \{1, \ldots, m\})\}$ *and*
   $[\![s]\!]^{G'} = [\![q^-]\!]^{G'} = \{(a_i, c_j) \mid i \neq j \in \{1, \ldots, m\})\} \cup \{(b_i, c_j) \mid i \neq j \in \{1, \ldots, m\})\}$.

5. $[\![s]\!]^G = [\![s]\!]^{G'} = C \times C$.

6. $[\![s]\!]^G = [\![s]\!]^{G'} = (A \cup B) \times (A \cup B)$.

7. $[\![s]\!]^G = [\![s]\!]^{G'} = id$.

8. $[\![s]\!]^G = [\![s]\!]^{G'} = \emptyset$.

Proof. We show this by systematically enumerating all strings until no new binary relations can be found. Note that we only enumerate over strings that alternate between property names and reversed property names. Indeed, all other strings evaluate to the empty relation on both G and G'. Every time we encounter new binary relations, we put the string in boldface.

s	$[\![s]\!]^G$	$[\![s]\!]^{G'}$
$\mathbf{id}$	id	id
$\mathbf{p}$	$C \times (A \cup B)$	$C \times (A \cup B)$
$\mathbf{q}$	$(C \times A) \cup$ $\{(c_i, b_j) \mid i \neq j \in \{1, \ldots, m\})\}$	$\{(c_i, a_j) \mid i \neq j \in \{1, \ldots, m\}\} \cup$ $\{(c_i, b_j) \mid i \neq j \in \{1, \ldots, m\}\}$
$\mathbf{p}^-$	$(A \cup B) \times C$	$(A \cup B) \times C$
$\mathbf{q}^-$	$(A \times C) \cup$ $\{(b_i, c_j) \mid i \neq j \in \{1, \ldots, m\})\}$	$\{(a_i, c_j) \mid i \neq j \in \{1, \ldots, m\})\} \cup$ $\{(b_i, c_j) \mid i \neq j \in \{1, \ldots, m\})\}$
$\mathbf{p/p^-}$	$C \times C$	$C \times C$
p/q^-	$C \times C$	$C \times C$
q/p^-	$C \times C$	$C \times C$
q/q^-	$C \times C$	$C \times C$
$\mathbf{p^-/p}$	$(A \cup B) \times (A \cup B)$	$(A \cup B) \times (A \cup B)$
p^-/q	$(A \cup B) \times (A \cup B)$	$(A \cup B) \times (A \cup B)$
q^-/p	$(A \cup B) \times (A \cup B)$	$(A \cup B) \times (A \cup B)$
q^-/q	$(A \cup B) \times (A \cup B)$	$(A \cup B) \times (A \cup B)$
$p/p^-/p$	$C \times (A \cup B)$	$C \times (A \cup B)$
$p/p^-/q$	$C \times (A \cup B)$	$C \times (A \cup B)$
$p^-/p/p^-$	$(A \cup B) \times C$	$(A \cup B) \times C$
$p^-/p/q^-$	$(A \cup B) \times C$	$(A \cup B) \times C$

□

We are now ready to prove the key proposition.

Proposition 4.29. *Let Σ be a vocabulary. Let $m \geq 3$ be a natural number. Let $V = A \cup B \cup C$ be the common set of nodes of the graphs $G = G_{\textit{full-eq}}(\Sigma, m)$ and $G' = G'_{\textit{full-eq}}(\Sigma, m)$. For all shapes φ over Σ counting to at most $m - 1$, we have $[\![\varphi]\!]^G = [\![\varphi]\!]^{G'}$. Moreover,*

- $[\![\varphi]\!]^G \cap V = A \cup B$, *or*
- $[\![\varphi]\!]^G \cap V = C$, *or*
- $[\![\varphi]\!]^G \cap V = V$, *or*
- $[\![\varphi]\!]^G \cap V = \emptyset$.

Proof. By induction on the structure of φ. For the base cases, if φ is $\top$ then $[\![\top]\!]^G = [\![\top]\!]^{G'} = N$ and $N \cap V = V$. If φ is $\mathit{hasValue}(c)$, then $[\![\mathit{hasValue}(c)]\!]^G = [\![\mathit{hasValue}(c)]\!]^{G'} = \{c\}$ and $\{c\} \cap V = \emptyset$ since $c \in \Sigma$ and $V \cap \Sigma = \emptyset$.

If φ is $\mathit{closed}(Q)$, we consider the possibilities for Q. If Q does not contain both p and q, then clearly $[\![\varphi]\!]^G \cap V = [\![\varphi]\!]^{G'} \cap V = A \cup B$. Otherwise, $[\![\varphi]\!]^G = [\![\varphi]\!]^{G'} = N$.

Before considering the remaining cases, we observe the following symmetries:

- All elements of A are symmetrical in G. This is obvious from the definition of G.
- Also in G', all elements of A are symmetrical. Indeed, for any $a_i \neq a_j$ in A, the function that swaps a_i and a_j, as well as c_i and c_j, is an automorphism of G'.
- Similarly, all elements of B are symmetrical in G, and also in G'.
- Moreover, we see that all elements of C are symmetrical in G, and in G'.
- Finally, in G', any a_i and b_j are symmetrical. Indeed, the function that swaps a_i and b_i is clearly an automorphism of G'. In turn, b_i and b_j are symmetrical by the above.

Therefore, we are only left to show:

(i) For any $a \in A$ and $b \in B$, we have $G, a \models \varphi \iff G, b \models \varphi$,

(ii) For any $a \in A$, we have $G, a \models \varphi \iff G', a \models \varphi$, and

(iii) For any $c \in C$, we have $G, c \models \varphi \iff G', c \models \varphi$.

(iv) For any $x \notin V$, we have $G, x \models \varphi \iff G', x \models \varphi$.

Note that then also for any $b \in B$, we have $G, b \models \varphi \iff G', b \models \varphi$ because for any $a \in A$ and $b \in B$, we have $G, b \models \varphi \overset{\text{(i)}}{\iff} G, a \models \varphi \overset{\text{(ii)}}{\iff} G', a \models \varphi \overset{\mathit{symmetry}}{\iff} G', b \models \varphi$.

Consider the case where φ is $\mathit{eq}(E, r)$. We verify (i), (ii), (iii), and (iv).

(i) By definition of G, $[\![r]\!]^G(a) = [\![r]\!]^G(b) = \emptyset$ for any property name r. Therefore we need to show $[\![E]\!]^G(a) = \emptyset \iff [\![E]\!]^G(b) = \emptyset$. By Lemma 4.7 we know there is a set of strings U equivalent to E in both G and G'. By Lemma 4.28 there are only 8 types of strings. We observe from Lemma 4.28 that for every U, $\bigcup_{s \in U} [\![s]\!]^G(a)$ is empty whenever U only contains strings of type 1, 2, 5, or 8. These are also exactly the U s.t. $\bigcup_{s \in U} [\![s]\!]^G(b)$ is empty.

(ii) Furthermore, these are also exactly the sets of strings U s.t. $\bigcup_{s \in U} [\![s]\!]^{G'}(a)$ is empty. Therefore, as $[\![r]\!]^{G'}(a) = \emptyset$, we have $G', a \models \varphi$.

(iii) Assume $G, c \models \varphi$. We consider the possibilities for r. First, suppose $r = p$. The sets of strings U s.t. $\bigcup_{s \in U} [\![s]\!]^G(c) = [\![p]\!]^G(c)$ contain strings of type 1 but not strings of type 5 or 7. These are also exactly the U s.t. $\bigcup_{s \in U} [\![s]\!]^{G'}(c) = [\![p]\!]^{G'}(c)$.

Next, suppose $r = q$. The sets of strings U s.t. $\bigcup_{s \in U} [\![s]\!]^G(c) = [\![p]\!]^G(c)$ contain strings of type 2 but not strings of type 1, 5 or 7. These are also exactly the types of strings s.t. $\bigcup_{s \in U} [\![s]\!]^{G'}(c) = [\![q]\!]^{G'}(c)$.

Finally, if r is any other property name, then $[\![r]\!]^G(c) = [\![r]\!]^{G'}(c) = \emptyset$. This is the case when U does not contain any strings of type 1, 2, 3, or 7. These are also exactly the types of strings U s.t. $\bigcup_{s \in U} [\![s]\!]^{G'}(c) = \emptyset$.

(iv) Let $x \in N - V$. Clearly $[\![r]\!]^G(x) = [\![r]\!]^{G'}(x) = \emptyset$. By Lemma 4.5, if E is safe, then $[\![E]\!]^G(x) = [\![E]\!]^{G'}(x) = \emptyset$. Therefore $G, x \models \varphi$ and $G', x \models \varphi$. On the other hand, whenever E is unsafe, $[\![E]\!]^G(x) = [\![E]\!]^{G'}(x) = \{x\} \neq \emptyset$. Therefore, $G, x \not\models \varphi$ and $G', x \not\models \varphi$.

Next, consider the case where φ is $disj(E_1, E_2)$. We again verify (i), (ii), (iii), and (iv).

(i) Assume $G, a \models disj(E_1, E_2)$. This can only be the case when the corresponding sets of strings U_1 and U_2 are of the following form. U_1 can consist only of strings of type 3, 4, 1, 2, 5, and 8 (Here, types 1, 2, 5 and 8 evaluate to empty from a as already seen above). U_2 can then only consist of strings of type 6, 7, 1, 2, 5, and 8 (or vice versa). These are also the only cases where $G, b \models disj(E_1, E_2)$.

(ii) Exactly the same situation occurs in G' and these are then also the only cases where $G', a \models disj(E_1, E_2)$.

(iii) Assume $G, c \models disj(E_1, E_2)$. This can only be the case when the corresponding sets of strings U_1 and U_2 are of the following form. U_1 can consist only of strings of type 1, 2, 3, 4, 6, and 8 (Here, types 3, 4, 6, and 8 evaluate to empty from c as already seen above). U_2 can then only consist of strings of type 5, 7, 3, 4, 6, and 8. We observe that this is also the case in G'.

(iv) Let $x \in N - V$. Whenever E_1 is safe, by Lemma 4.5 $[\![E_1]\!]^G(x) = [\![E_1]\!]^{G'}(x) = \emptyset$. Therefore, $G, x \models \varphi$ and $G', x \models \varphi$. Clearly, the same holds whenever E_2 is safe. When both E_1 and E_2 are unsafe, $[\![E_1]\!]^G(x) = [\![E_1]\!]^{G'}(x) = \{x\} \neq \emptyset$. Therefore, $G, x \not\models \varphi$ and $G', x \not\models \varphi$.

The cases where φ is $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ or $\neg\varphi_1$ are handled by induction in a straightforward manner.

Lastly, we consider the case where φ is $\geq_n E.\varphi_1$.

(i) Assume $G, a \models \varphi$. Then, there exist distinct $x_1, \dots, x_n$ s.t. $(a, x_i) \in [\![E]\!]^G$ and $G, x_i \models \varphi_1$ for $1 \leq i \leq n$. Again, by Lemma 4.7 we know there is a set of strings U equivalent to E in both G and G'. By Lemma 4.28 there are only 8 types of strings. By induction, we consider three cases.

First, if $[\![\varphi_1]\!]^G \cap V = A \cup B$, then $x_i, \dots, x_n \in A \cup B$. Therefore, we know U must at least contain strings of type 6 or 7. Suppose U contains strings of type 6. Then, we verify that $\sharp(\bigcup_{s \in U} [\![s]\!]^G(b) \cap (A \cup B)) \geq m - 1 \geq n$. Whenever U contains strings of type 7, and not of type 6, we know $n = 1$ and clearly $\sharp(\bigcup_{s \in U} [\![s]\!]^G(b) \cap (A \cup B)) = 1$.

Next, if $[\![\varphi_1]\!]^G \cap V = C$, then $x_i, \dots, x_n \in C$. Therefore, we know U must at least contain strings of type 3 or 4. Whenever U contains 3 or 4, we verify that $\sharp(\bigcup_{s \in U} [\![s]\!]^G(b) \cap C) \geq m - 1 \geq n$.

Next, if $[\![\varphi_1]\!]^G \cap V = V$, then $x_i, \dots, x_n \in V$. Therefore, we know U must at least contain strings of type 3, 4, 6 or 7. All these types have already been handled in the previous two cases.

Finally, the case where $[\![\varphi_1]\!]^G \cap V = \emptyset$ cannot occur as $n > 0$.

The sets of strings U above are also exactly the sets used to argue the implication from right to left.

(ii) For every case of U above, for every inductive case of φ_1, we can also verify that $\sharp(\bigcup_{s \in U} [\![s]\!]^{G'}(a) \cap (A \cup B)) \geq m - 1 \geq n$, $\sharp(\bigcup_{s \in U} [\![s]\!]^{G'}(a) \cap C) \geq m - 1 \geq n$, and $\sharp(\bigcup_{s \in U} [\![s]\!]^{G'}(a) \cap V) \geq m - 1 \geq n$.

(iii) Assume $G, c \models \varphi$. Then, there exist distinct $x_1, \ldots, x_n$ s.t. $(c, x_i) \in [\![E]\!]^G$ and $G, x_i \models \varphi_1$ for $1 \leq i \leq n$. By induction, we consider three cases.

First, if $[\![\varphi_1]\!]^G \cap V = A \cup B$, then $x_i, \ldots, x_n \in A \cup B$. Therefore, we know U must at least contain strings of type 1 or 2. Whenever U contains 1 or 2, we verify that $\sharp(\bigcup_{s \in U} [\![s]\!]^{G'}(c) \cap (A \cup B)) \geq m \geq n$.

Next, if $[\![\varphi_1]\!]^G \cap V = C$, then $x_i, \ldots, x_n \in C$. Therefore, we know U must at least contain strings of type 5 or 7. Whenever U contains 5, we verify that $\sharp(\bigcup_{s \in U} [\![s]\!]^{G'}(c) \cap C) \geq m \geq n$. Otherwise, whenever U contains strings of type 7, and not of type 5, we know $n = 1$ and clearly $\sharp(\bigcup_{s \in U} [\![s]\!]^{G'}(c) \cap C) = 1$.

Next, if $[\![\varphi_1]\!]^G \cap V = V$, then $x_i, \ldots, x_n \in V$. Therefore, we know U must at least contain strings of type 1, 2, 5 or 7. All these types have already been handled in the previous two cases.

Finally, the case where $[\![\varphi_1]\!]^G \cap V = \emptyset$ cannot occur as $n > 0$.

The sets of strings U above are also exactly the sets used to argue the implication from right to left.

(iv) For the direction from left to right, take $x \in [\![\varphi]\!]^G \setminus V$. Since $G, x \models \varphi$, there exists $y_1, \ldots, y_n$ s.t. $(x, y_i) \in [\![E]\!]^G$ and $G, y_i \models \varphi_1$ for $i = 1, \ldots, n$. However, since $x \notin V$, by Lemma 4.5, all y_i must equal x. Hence, $n = 1$ and $(x, x) \in [\![E]\!]^G$ and $G, x \models \varphi_1$. Then again, by the same Lemma, $(x, x) \in [\![E]\!]^{G'}$, since G and G' have the same set of nodes V. Moreover, by induction, $G', x \models \varphi_1$. We conclude $G', x \models \varphi$ as desired. The direction from right to left is argued symmetrically. □

4.3.2 Full disjointness

We present here the proof for the primitivity of full disjointness tests. The general strategy is the same as in Section 4.3.1.

We begin by defining the graphs G and G' formally.

Definition 4.30. $G_{\textit{full-disj}}(\Sigma, m)$ Let m be a natural number that is a multiple of 8. Let $A = \{a_1, \ldots, a_m\}$, $B = \{b_1, \ldots, b_m\}$ and $C = \{c_1, \ldots, c_m\}$ be three disjoint sets of nodes, disjoint from Σ. For any $i \leq j$, we write $a_{i \to j}$ to denote the set

$$\{a_{1+(i-1+l \mod m)} \mid 0 \leq l \leq j - i\}$$

We define $b_{i \to j}$ and $c_{i \to j}$ analogously.

We define the graph $G_{\textit{full-disj}}(\Sigma, m)$ by:

$$[\![p]\!]^G(c_i) = a_{i \to i + \frac{m}{2} - 1} \cup b_{i - \frac{m}{8} \to i + \frac{m}{2} - 1} \text{ and}$$
$$[\![q]\!]^G(c_i) = a_{i - \frac{m}{2} \to i - 1} \cup b_{i - \frac{m}{2} \to i + \frac{m}{8} - 1} \text{ for } 1 \leq i \leq m$$

The p and q relations are visualized in Figure 4.2.

To give an example for our notation, suppose $m = 8$. Then,

$$a_{2 \to 5} = \{a_{1+(1+l \mod 8)} \mid 0 \leq l \leq 3\} = \{a_2, a_3, a_4, a_5\}$$
$$a_{7 \to 10} = \{a_{1+(6+l \mod 8)} \mid 0 \leq l \leq 3\} = \{a_7, a_8, a_1, a_2\}, \text{ and}$$
$$a_{-4 \to -1} = \{a_{1+(-5+l \mod 8)} \mid 0 \leq l \leq 3\} = \{a_4, a_5, a_6, a_7\}$$

Definition 4.31. $G'_{\textit{full-disj}}(\Sigma, m)$ We define the graph $G'_{\textit{full-disj}}(\Sigma, m)$ on the same nodes as $G_{\textit{full-disj}}(\Sigma, m)$, with the only difference being the relationship of the p- and q-edges from C to A:

$$\llbracket p \rrbracket^G(c_i) = a_{i-\frac{m}{8}\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1} \text{ and}$$
$$\llbracket q \rrbracket^G(c_i) = a_{i-\frac{m}{2}\to i+\frac{m}{8}-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1} \text{ for } 1 \leq i \leq m$$

The p and q relations are visualized in Figure 4.2.

Important to the intuition behind these graphs is the overlap generated by the inverse p- and q-edges. As demonstrated in Figure 4.3, in graph $G = G_{\textit{full-disj}}(\Sigma, m)$, the set of c nodes reached from a nodes with inverse p edges is disjoint from the set of c nodes reached with inverse q edges. This is not the case for b nodes: there, these sets overlap by precisely one fourth of the c nodes. For graph $G' = G'_{\textit{full-disj}}(\Sigma, m)$, the sets of c nodes reachable by inverse p and q edges overlap for both a and b nodes.

We precisely characterize the behavior of strings on the graphs G and G' as follows.

Lemma 4.32. *Let Σ be a vocabulary. Let m be a natural number that is a multiple of 8. Let G be $G_{\textit{full-disj}}(\Sigma, m)$ and let G' be $G'_{\textit{full-disj}}(\Sigma, m)$. The only possibilities for a string s evaluated on G and G' are the following:*

1. $\llbracket s \rrbracket^G = \llbracket p \rrbracket^G = \bigcup_{i\in\{1,\ldots,m\}}\{c_i\} \times (a_{i\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1})$ *and*
   $\llbracket s \rrbracket^{G'} = \llbracket p \rrbracket^{G'} = \bigcup_{i\in\{1,\ldots,m\}}\{c_i\} \times (a_{i-\frac{m}{8}\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1})$*;*

2. $\llbracket s \rrbracket^G = \llbracket q \rrbracket^G = \bigcup_{i\in\{1,\ldots,m\}}\{c_i\} \times (a_{i-\frac{m}{2}\to i-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1})$ *and*
   $\llbracket s \rrbracket^{G'} = \llbracket q \rrbracket^{G} = \bigcup_{i\in\{1,\ldots,m\}}\{c_i\} \times (a_{i-\frac{m}{2}\to i+\frac{m}{8}-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1})$*;*

3. $\llbracket s \rrbracket^G = \llbracket p^- \rrbracket^G = \bigcup_{i\in\{1,\ldots,m\}}(\{a_i\} \times c_{i-\frac{m}{2}+1\to i}) \cup (\{b_i\} \times c_{i-\frac{m}{2}+1\to i+\frac{m}{8}})$ *and*
   $\llbracket s \rrbracket^{G'} = \llbracket p^- \rrbracket^{G'} = \bigcup_{i\in\{1,\ldots,m\}}(\{a_i\} \times c_{i-\frac{m}{2}+1\to i+\frac{m}{8}}) \cup (\{b_i\} \times c_{i-\frac{m}{2}+1\to i+\frac{m}{8}})$ *;*

4. $\llbracket s \rrbracket^G = \llbracket q^- \rrbracket^G = \bigcup_{i\in\{1,\ldots,m\}}(\{a_i\} \times c_{i+1\to i+\frac{m}{2}}) \cup (\{b_i\} \times c_{i-\frac{m}{8}+1\to i+\frac{m}{2}})$ *and*
   $\llbracket s \rrbracket^{G'} = \llbracket q^- \rrbracket^{G'} = \bigcup_{i\in\{1,\ldots,m\}}(\{a_i\} \times c_{i-\frac{m}{8}+1\to i+\frac{m}{2}}) \cup (\{b_i\} \times c_{i-\frac{m}{8}+1\to i+\frac{m}{2}})$*;*

5. $\llbracket s \rrbracket^G = \llbracket s \rrbracket^{G'} = C \times C$*;*

6. $\llbracket s \rrbracket^G = \llbracket s \rrbracket^{G'} = (A \cup B) \times (A \cup B)$*;*

7. $\llbracket s \rrbracket^G = \llbracket s \rrbracket^{G'} = C \times (A \cup B)$*;*

8. $\llbracket s \rrbracket^G = \llbracket s \rrbracket^{G'} = (A \cup B) \times C$*;*

9. $\llbracket s \rrbracket^G = \llbracket s \rrbracket^{G'} = id$*; or*

10. $\llbracket s \rrbracket^G = \llbracket s \rrbracket^{G'} = \emptyset$

The first four types of strings are visualized in Figure 4.2 and Figure 4.3.

Proof. The proof is performed as in the proof of Lemma 4.28. We now have the following table:

s	$[\![s]\!]^G$	$[\![s]\!]^{G'}$
$\mathbf{id}$	id	id
$\mathbf{p}$	type 1	type 1
$\mathbf{q}$	type 2	type 2
$\mathbf{p}^-$	type 3	type 3
$\mathbf{q}^-$	type 4	type 4
$\mathbf{p}/\mathbf{p}^-$	$C \times C$	$C \times C$
p/q^-	$C \times C$	$C \times C$
q/p^-	$C \times C$	$C \times C$
q/q^-	$C \times C$	$C \times C$
$\mathbf{p}^-/\mathbf{p}$	$(A \cup B) \times (A \cup B)$	$(A \cup B) \times (A \cup B)$
p^-/q	$(A \cup B) \times (A \cup B)$	$(A \cup B) \times (A \cup B)$
q^-/p	$(A \cup B) \times (A \cup B)$	$(A \cup B) \times (A \cup B)$
q^-/q	$(A \cup B) \times (A \cup B)$	$(A \cup B) \times (A \cup B)$
$\mathbf{p}/\mathbf{p}^-/\mathbf{p}$	$C \times (A \cup B)$	$C \times (A \cup B)$
$p/p^-/q$	$C \times (A \cup B)$	$C \times (A \cup B)$
$\mathbf{p}^-/\mathbf{p}/\mathbf{p}^-$	$(A \cup B) \times C$	$(A \cup B) \times C$
$p^-/p/q^-$	$(A \cup B) \times C$	$(A \cup B) \times C$
$p/p^-/p/p^-$	$C \times C$	$C \times C$
$p/p^-/p/q^-$	$C \times C$	$C \times C$
$p^-/p/p^-/p$	$(A \cup B) \times (A \cup B)$	$(A \cup B) \times (A \cup B)$
$p^-/p/p^-/q$	$(A \cup B) \times (A \cup B)$	$(A \cup B) \times (A \cup B)$

□

We are ready to present our key Proposition.

Proposition 4.33. *Let Σ be a vocabulary. Let m be a natural number and a multiple of 8. Let $V = A \cup B \cup C$ be the common set of nodes of the graphs $G = G_{\mathit{full\text{-}disj}}(\Sigma, m)$ and $G' = G'_{\mathit{full\text{-}disj}}(\Sigma, m)$. For all shapes φ over Σ counting to at most $\frac{m}{2}$, we have $[\![\varphi]\!]^G = [\![\varphi]\!]^{G'}$. Moreover,*

- $[\![\varphi]\!]^G \cap V = A \cup B$, *or*
- $[\![\varphi]\!]^G \cap V = C$, *or*
- $[\![\varphi]\!]^G \cap V = V$, *or*
- $[\![\varphi]\!]^G \cap V = \emptyset$.

Proof. By induction on the structure of φ. For the base cases, if φ is $\top$ then $[\![\top]\!]^G = [\![\top]\!]^{G'} = N$ and $N \cap V = V$. If φ is $\mathit{hasValue}(c)$, then $[\![\mathit{hasValue}(c)]\!]^G = [\![\mathit{hasValue}(c)]\!]^{G'} = \{c\}$ and $\{c\} \cap V = \emptyset$ since $c \in \Sigma$ and $V \cap \Sigma = \emptyset$.

If φ is $\mathit{closed}(Q)$, we consider the possibilities for Q. If Q does not contain both p and q, then clearly $[\![\varphi]\!]^G \cap V = [\![\varphi]\!]^{G'} \cap V = A \cup B$. Otherwise, $[\![\varphi]\!]^G = [\![\varphi]\!]^{G'} = N$.

Before considering the remaining cases, we observe the following symmetries:

- In both G and G', all elements of A are symmetrical, as are all elements of B, and all elements of C. Indeed, for any $i \in \{1, \ldots, m\}$, the function that maps x_i to $x_{1+i \mod m}$ where x_i is a_i, b_i or c_i, is clearly an automorphism of G and also of G'.
- Furthermore, in G', any a_i and b_j are symmetrical. Indeed, the function that swaps every a_i with b_i is an automorphism of G'. (We already know that b_i and b_j are symmetrical by the above.)

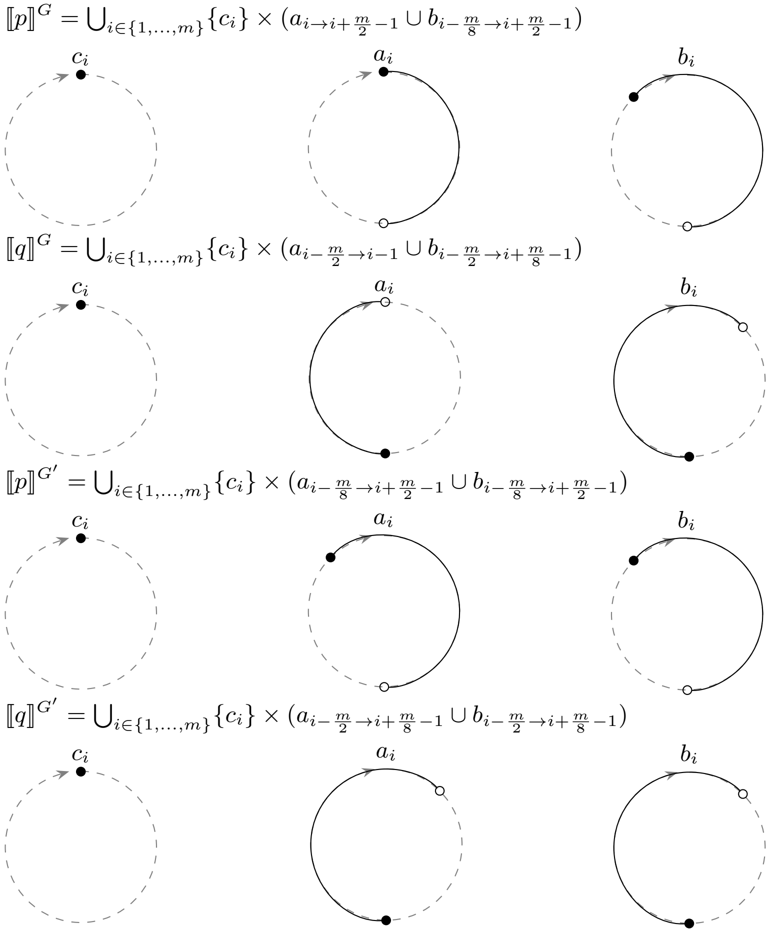
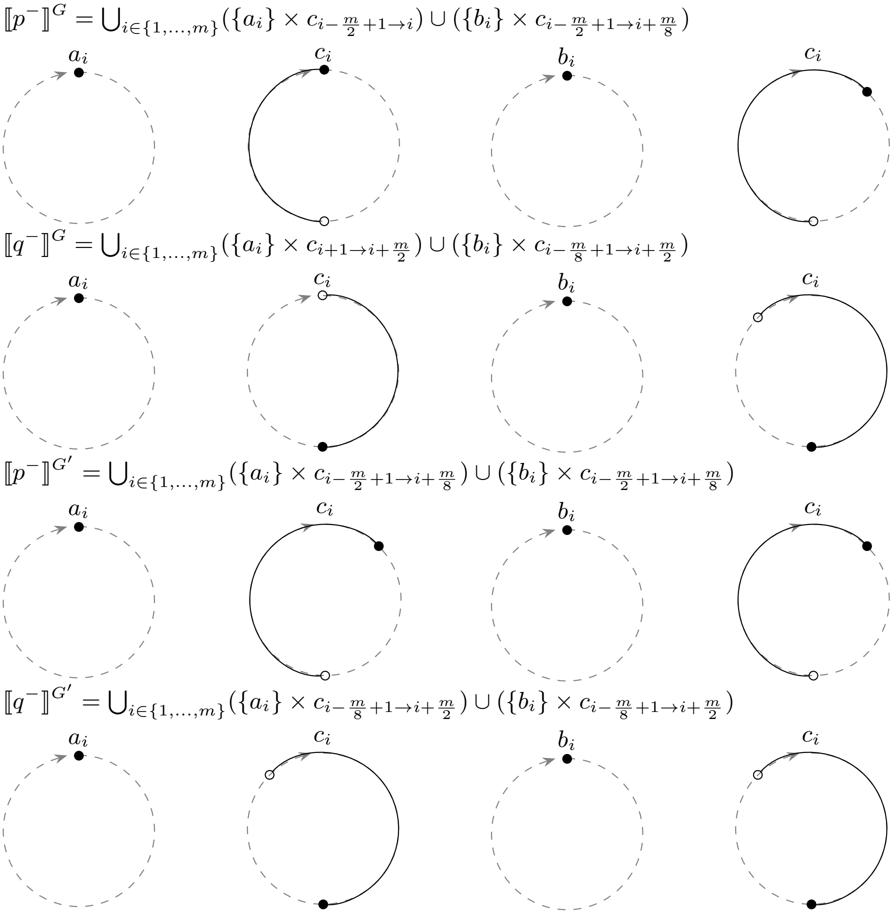

$[\![p]\!]^G = \bigcup_{i\in\{1,\ldots,m\}}\{c_i\} \times (a_{i\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1})$
c_i
a_i
b_i
$[\![q]\!]^G = \bigcup_{i\in\{1,\ldots,m\}}\{c_i\} \times (a_{i-\frac{m}{2}\to i-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1})$
c_i
a_i
b_i
$[\![p]\!]^{G'} = \bigcup_{i\in\{1,\ldots,m\}}\{c_i\} \times (a_{i-\frac{m}{8}\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1})$
c_i
a_i
b_i
$[\![q]\!]^{G'} = \bigcup_{i\in\{1,\ldots,m\}}\{c_i\} \times (a_{i-\frac{m}{2}\to i+\frac{m}{8}-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1})$
c_i
a_i
b_i

Figure 4.2: Illustration of the p and q relations in graphs $G = G_{\mathit{full\text{-}disj}}(\Sigma, m)$ and $G' = G'_{\mathit{full\text{-}disj}}(\Sigma, m)$



$[\![p^-]\!]^G = \bigcup_{i\in\{1,\ldots,m\}}(\{a_i\} \times c_{i-\frac{m}{2}+1\to i}) \cup (\{b_i\} \times c_{i-\frac{m}{2}+1\to i+\frac{m}{8}})$

$[\![q^-]\!]^G = \bigcup_{i\in\{1,\ldots,m\}}(\{a_i\} \times c_{i+1\to i+\frac{m}{2}}) \cup (\{b_i\} \times c_{i-\frac{m}{8}+1\to i+\frac{m}{2}})$

$[\![p^-]\!]^{G'} = \bigcup_{i\in\{1,\ldots,m\}}(\{a_i\} \times c_{i-\frac{m}{2}+1\to i+\frac{m}{8}}) \cup (\{b_i\} \times c_{i-\frac{m}{2}+1\to i+\frac{m}{8}})$

$[\![q^-]\!]^{G'} = \bigcup_{i\in\{1,\ldots,m\}}(\{a_i\} \times c_{i-\frac{m}{8}+1\to i+\frac{m}{2}}) \cup (\{b_i\} \times c_{i-\frac{m}{8}+1\to i+\frac{m}{2}})$

Figure 4.3: Illustration of the p^- and q^- relations in graphs $G = G_{\textit{full-disj}}(\Sigma, m)$ and $G' = G'_{\textit{full-disj}}(\Sigma, m)$

Therefore, we are only left to show:

(i) For any $a \in A$ and $b \in B$, we have $G, a \models \varphi \iff G, b \models \varphi$,

(ii) For any $a \in A$, we have $G, a \models \varphi \iff G', a \models \varphi$, and

(iii) For any $c \in C$, we have $G, c \models \varphi \iff G', c \models \varphi$.

(iv) For any $x \notin V$, we have $G, x \models \varphi \iff G', x \models \varphi$.

Note that then also for any $b \in B$, we have $G, b \models \varphi \iff G', b \models \varphi$ because for any $a \in A$ and $b \in B$, we have $G, b \models \varphi \overset{\text{(i)}}{\iff} G, a \models \varphi \overset{\text{(ii)}}{\iff} G', a \models \varphi \overset{\textit{symmetry}}{\iff} G', b \models \varphi$.

Consider the case where φ is $\mathit{disj}(E, r)$. We verify (ii), (ii), (iii), and (iv). First, to see that (ii) and (ii) hold, we observe that $[\![r]\!]^G(a) = [\![r]\!]^G(b) = [\![r]\!]^{G'}(a) = [\![r]\!]^{G'}(b) = \emptyset$. Therefore, $G, a \models \varphi$, $G, b \models \varphi$, $G', a \models \varphi$, and $G', b \models \varphi$ always hold, showing (ii) and (ii).

Next, to show (iii) where $r = p$, assume $G, c \models \mathit{disj}(E, p)$. By Lemma 4.7 we know there is a set of strings U equivalent to E in both G and G'. By Lemma 4.32 there are only 10 types of strings. We observe from Lemma 4.32 that for every U, $\bigcup_{s \in U} [\![s]\!]^G(c)$ is disjoint from $[\![p]\!]^G(c)$ whenever U does not contain strings of type 1, 2, or 7. These are also exactly the U s.t. $\bigcup_{s \in U} [\![s]\!]^{G'}(c)$ is disjoint from $[\![p]\!]^{G'}(c)$.

Next, to show (iii) where $r = q$, assume $G, c \models \mathit{disj}(E, q)$. We observe from Lemma 4.32 that for every U, $\bigcup_{s \in U} [\![s]\!]^G(c)$ is disjoint from $[\![q]\!]^G(c)$ whenever U does not contain strings of type 1, 2, or 7. These are also exactly the U s.t. $\bigcup_{s \in U} [\![s]\!]^{G'}(c)$ is disjoint from $[\![q]\!]^{G'}(c)$.

For every other property name r, $[\![r]\!]^G(a) = [\![r]\!]^G(b) = \emptyset$. Therefore, $G, c \models \varphi$ and $G', c \models \varphi$ always hold.

Finally, we show (iv) by observing that for any $x \in N \setminus V$, $[\![r]\!]^G(x) = [\![r]\!]^{G'}(x) = \emptyset$. Therefore, $G, x \models \varphi$ and $G', x \models \varphi$ always hold.

Next, consider the case where φ is $\mathit{eq}(E_1, E_2)$. We again verify (ii), (ii), (iii), and (iv).

We show (ii) by using a canonical labeling argument. For any two sets U_1 and U_2 of types, we call U_1 and U_2 equivalent in $a \in A$ if $\bigcup_{s \in U_1} [\![s]\!]^G(a) = \bigcup_{s \in U_2} [\![s]\!]^G(a)$. Similarly, we define when U_1 and U_2 are equivalent in b or c.

We can canonically label the equivalence classes in a as follows. Let $U = \{u_1, \dots, u_l\}$ and $u_1 < \cdots < u_l$ with each $u_i \in \{1, \dots, 10\}$ a type.

There are only six unique singleton sets namely $\{1\}$, $\{3\}$, $\{4\}$, $\{6\}$, $\{8\}$, and $\{9\}$. Replace each u_i by their singleton representative $\overline{u_i}$. In $\{\overline{u_1}, \dots, \overline{u_l}\}$, reorder and remove duplicates to obtain an equivalent set $\{u'_1, \dots, u'_{l'}\}$.

If $l' = 1$, we are done. Otherwise, we enumerate all nonequivalent 2-element sets that are not equivalent to a singleton: there are again six of those, namely $\{3, 6\}$, $\{3, 9\}$, $\{4, 6\}$, $\{4, 9\}$, $\{6, 8\}$, and $\{8, 9\}$.

Replace u'_1 and u'_2 by either $\{u''\}$ in case $\{u'_1, u'_2\}$ is equivalent to a singleton; otherwise replace u'_1 and u'_2 by their equivalent 2-element set $\{u''_1, u''_2\}$. If $l' = 2$, we are again done.

We can repeat this process. However, it turns out that there are no 3-element sets that are not equivalent to a singleton or a 2-element set. Hence, there are only 12 representatives. The enumeration process is shown in Table 4.1, giving the representative for equivalence in a as well as for equivalence in b. Crucially, in filling the table, we observe every set U has the same representative for equivalence in a as for equivalence in b.

Next, (ii) is shown in an analogous manner, where the enumeration process is again shown in Table 4.1.

Next, (iii) is again shown with an analogous manner, where the enumeration process is shown in Table 4.2.

To show (iv), assume $x \notin V$. If both E_1 and E_2 are safe, then by Lemma 4.5 $[\![E_1]\!]^G(x) = [\![E_2]\!]^{G'}(x) = \emptyset$. Thus, $G, x \models \varphi$ and $G', x \models \varphi$. If both E_1 and E_2 are unsafe, then by Lemma

4.5 $[\![E_1]\!]^G(x) = [\![E_2]\!]^{G'}(x) = \{x\}$. Thus, $G, x \models \varphi$ and $G', x \models \varphi$. However, whenever only one of E_1 and E_2 is safe, clearly $G, x \not\models \varphi$ and $G', x \not\models \varphi$.

The cases where φ is $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ or $\neg\varphi_1$ are handled by induction in a straightforward manner.

Lastly, we consider the case where φ is $\geq_n E.\psi$.

(ii) Assume $G, a \models \varphi$. Then, there exist distinct $x_1, \ldots, x_n$ s.t. $(a, x_i) \in [\![E]\!]^G$ and $G, x_i \models \psi$ for $1 \leq i \leq n$. Again, by Lemma 4.7 we know there is a set of strings U equivalent to E in both G and G'. By Lemma 4.32 there are only 10 types of strings. By induction, we consider three cases.

First, if $[\![\psi]\!]^G \cap V = A \cup B$, then $x_i, \ldots, x_n \in A \cup B$. Therefore, we know U must at least contain strings of type 6 or 9. Suppose U contains strings of type 6. Then, we verify that $\sharp(\bigcup_{s \in U} [\![s]\!]^G(b) \cap (A \cup B)) \geq m \geq n$. Otherwise, whenever U contains strings of type 9, and not of type 6, we know $n = 1$ and clearly $\sharp(\bigcup_{s \in U} [\![s]\!]^G(b) \cap (A \cup B)) = 1$.

Next, if $[\![\psi]\!]^G \cap V = C$, then $x_i, \ldots, x_n \in C$. Therefore, we know U must at least contain strings of type 3, 4 or 8. Whenever U contains 3, 4 or 8, we verify that $\sharp(\bigcup_{s \in U} [\![s]\!]^G(b) \cap C) \geq \frac{m}{2} \geq n$.

Next, if $[\![\psi]\!]^G \cap V = V$, then $x_i, \ldots, x_n \in V$. Therefore, we know U must at least contain strings of type 3, 4, 6, 8 or 9. All these types have already been handled in the previous two cases.

Finally, the case where $[\![\psi]\!]^G \cap V = \emptyset$ cannot occur as $n > 0$.

The sets of strings U above are also exactly the sets used to argue the implication from right to left.

(ii) For every case of U above, for every inductive case of ψ, we can also verify that $\sharp(\bigcup_{s \in U} [\![s]\!]^{G'}(a) \cap (A \cup B)) \geq m \geq n$, $\sharp(\bigcup_{s \in U} [\![s]\!]^{G'}(a) \cap C) \geq \frac{m}{2} \geq n$, and $\sharp(\bigcup_{s \in U} [\![s]\!]^{G'}(a) \cap V) \geq \frac{m}{2} \geq n$.

(iii) Assume $G, c \models \varphi$. Then, there exist distinct $x_1, \ldots, x_n$ s.t. $(c, x_i) \in [\![E]\!]^G$ and $G, x_i \models \psi$ for $1 \leq i \leq n$. By induction, we consider three cases.

First, if $[\![\psi]\!]^G \cap V = A \cup B$, then $x_i, \ldots, x_n \in A \cup B$. Therefore, we know U must at least contain strings of type 1, 2 or 7. Whenever U contains 1, 2 or 7, we verify that $\sharp(\bigcup_{s \in U} [\![s]\!]^{G'}(c) \cap (A \cup B)) \geq \frac{m}{2} \geq n$.

Next, if $[\![\psi]\!]^G \cap V = C$, then $x_i, \ldots, x_n \in C$. Therefore, we know U must at least contain strings of type 5 or 9. Whenever U contains 5, we verify that $\sharp(\bigcup_{s \in U} [\![s]\!]^{G'}(c) \cap C) \geq m \geq n$. Otherwise, whenever U contains strings of type 9, and not of type 5, we know $n = 1$ and clearly $\sharp(\bigcup_{s \in U} [\![s]\!]^{G'}(c) \cap C) = 1$.

Next, if $[\![\psi]\!]^G \cap V = V$, then $x_i, \ldots, x_n \in V$. Therefore, we know U must at least contain strings of type 1, 2, 5, 7 or 9. All these types have already been handled in the previous two cases.

Finally, the case where $[\![\psi]\!]^G \cap V = \emptyset$ cannot occur as $n > 0$.

The sets of strings U above are also exactly the sets used to argue the implication from right to left.

(iv) For the direction from left to right, take $x \in [\![\varphi]\!]^G \setminus V$. Since $G, x \models \varphi$, there exists $y_1, \ldots, y_n$ s.t. $(x, y_i) \in [\![E]\!]^G$ and $G, y_i \models \psi$ for $i = 1, \ldots, n$. However, since $x \notin V$, by Lemma 4.5, all y_i must equal x. Hence, $n = 1$ and $(x, x) \in [\![E]\!]^G$ and $G, x \models \psi$. Then again, by the same Lemma, $(x, x) \in [\![E]\!]^{G'}$, since G and G' have the same set of

Table 4.1: Sets of types starting from a_i, b_i in G and a_i in G'.

U	$[\![E]\!]^G(a_i)$	$[\![E]\!]^G(b_i)$	$[\![E]\!]^{G'}(a_i)$
$\{\mathbf{1}\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\{2\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\{\mathbf{3}\}$	$c_{i-\frac{k}{2}+1\rightarrow i}$	$c_{i-\frac{k}{2}+1\rightarrow i+\frac{k}{8}}$	$c_{i-\frac{k}{2}+1\rightarrow i+\frac{k}{8}}$
$\{\mathbf{4}\}$	$c_{i+1\rightarrow i+\frac{k}{2}}$	$c_{i-\frac{k}{8}+1\rightarrow i+\frac{k}{2}}$	$c_{i-\frac{k}{8}+1\rightarrow i+\frac{k}{2}}$
$\{5\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\{\mathbf{6}\}$	$A \cup B$	$A \cup B$	$A \cup B$
$\{7\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\{\mathbf{8}\}$	C	C	C
$\{\mathbf{9}\}$	$\{a_i\}$	$\{b_i\}$	$\{a_i\}$
$\{10\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\{3,4\}$	C	C	C
$\{\mathbf{3},\mathbf{6}\}$	$A \cup B \cup c_{i-\frac{k}{2}+1\rightarrow i}$	$A \cup B \cup c_{i-\frac{k}{2}+1\rightarrow i+\frac{k}{8}}$	$A \cup B \cup c_{i-\frac{k}{2}+1\rightarrow i+\frac{k}{8}}$
$\{3,8\}$	C	C	
$\{\mathbf{3},\mathbf{9}\}$	$c_{i+1\rightarrow i+\frac{k}{2}} \cup \{a_i\}$	$c_{i-\frac{k}{8}+1\rightarrow i+\frac{k}{2}} \cup \{b_i\}$	$c_{i-\frac{k}{8}+1\rightarrow i+\frac{k}{2}} \cup \{a_i\}$
$\{\mathbf{4},\mathbf{6}\}$	$A \cup B \cup c_{i+1\rightarrow i+\frac{k}{2}}$	$A \cup B \cup c_{i-\frac{k}{8}+1\rightarrow i+\frac{k}{2}}$	$A \cup B \cup c_{i-\frac{k}{8}+1\rightarrow i+\frac{k}{2}}$
$\{4,8\}$	C	C	
$\{\mathbf{4},\mathbf{9}\}$	$c_{i+1\rightarrow i+\frac{k}{2}} \cup \{a_i\}$	$c_{i-\frac{k}{8}+1\rightarrow i+\frac{k}{2}} \cup \{b_i\}$	$c_{i-\frac{k}{8}+1\rightarrow i+\frac{k}{2}} \cup \{a_i\}$
$\{\mathbf{6},\mathbf{8}\}$	V	V	V
$\{6,9\}$	$A \cup B$	$A \cup B$	$A \cup B$
$\{\mathbf{8},\mathbf{9}\}$	$C \cup \{a_i\}$	$C \cup \{b_i\}$	$C \cup \{a_i\}$
$\{3,6,4\}$	V	V	V
$\{3,6,8\}$	V	V	V
$\{3,6,9\}$	$A \cup B \cup c_{i-\frac{k}{2}+1\rightarrow i}$	$A \cup B \cup c_{i-\frac{k}{2}+1\rightarrow i+\frac{k}{8}}$	$A \cup B \cup c_{i-\frac{k}{2}+1\rightarrow i+\frac{k}{8}}$
$\{3,9,4\}$	$C \cup \{a_i\}$	$C \cup \{b_i\}$	$C \cup \{a_i\}$
$\{4,9,6\}$	$A \cup B \cup c_{i+1\rightarrow i+\frac{k}{2}}$	$A \cup B \cup c_{i-\frac{k}{8}+1\rightarrow i+\frac{k}{2}}$	$A \cup B \cup c_{i-\frac{k}{8}+1\rightarrow i+\frac{k}{2}}$
$\{4,9,8\}$	C	C	C
$\{8,9,3\}$	$C \cup \{a_i\}$	$C \cup \{b_i\}$	$C \cup \{a_i\}$
$\{4,6,3\}$	V	V	V

nodes $V/$ Moreover, by induction, $G', x \models \psi$. We conclude $G', x \models \varphi$ as desired. The direction from right to left is argued symmetrically. □

Remark 4.34. In our construction of the graphs G and G', we work with segments that overlap for 1/8th of the number of nodes. The critical reader will remark that an overlap of a single node would already be sufficient. Our choice for working with a larger overlap is indeed largely aesthetic. Moreover, our proof still works for an extension of SHACL where shapes of the form $|r \cap E| \geq n$ would be allowed. This extension allows us to write shapes like $|\mathsf{colleague} \cap \mathsf{friend}| \geq 5$, stating that the node has at least five colleagues that are also friends. Such an extension then would still not be able to express full disjointness.

4.3.3 Further non-definability results

In Theorem 4.1, we showed that equality is primitive in $\mathcal{L}(\mathit{disj}, \mathit{closed})$, and similarly, that disjointness is primitive in $\mathcal{L}(\mathit{eq}, \mathit{closed})$. Can we strengthen these results to $\mathcal{L}(\mathit{full\text{-}disj}, \mathit{closed})$ and $\mathcal{L}(\mathit{full\text{-}eq}, \mathit{closed})$, respectively? This turns out to be indeed possible.

Table 4.2: Sets of types starting from c_i in G and in G'.

U	$[\![E]\!]^G(c_i)$	$[\![E]\!]^{G'}(c_i)$
$\{\mathbf{1}\}$	$a_{i\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1}$	$a_{i-\frac{m}{8}\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1}$
$\{\mathbf{2}\}$	$a_{i-\frac{m}{2}\to i-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1}$	$a_{i-\frac{m}{2}\to i+\frac{m}{8}-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1}$
$\{\mathbf{3}\}$	$\emptyset$	$\emptyset$
$\{4\}$	$\emptyset$	$\emptyset$
$\{\mathbf{5}\}$	C	C
$\{6\}$	$\emptyset$	$\emptyset$
$\{\mathbf{7}\}$	$A \cup B$	$A \cup B$
$\{8\}$	$\emptyset$	$\emptyset$
$\{\mathbf{9}\}$	$\{c_i\}$	$\{c_i\}$
$\{10\}$	$\emptyset$	$\emptyset$
$\{1,2\}$	$A \cup B$	$A \cup B$
$\{\mathbf{1},\mathbf{5}\}$	$C \cup a_{i\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1}$	$C \cup a_{i-\frac{m}{8}\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1}$
$\{1,7\}$	$A \cup B$	$A \cup B$
$\{\mathbf{1},\mathbf{9}\}$	$\{c_i\} \cup a_{i\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1}$	$\{c_i\} \cup a_{i-\frac{m}{8}\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1}$
$\{\mathbf{2},\mathbf{5}\}$	$C \cup a_{i-\frac{m}{2}\to i-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1}$	$C \cup a_{i-\frac{m}{2}\to i+\frac{m}{8}-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1}$
$\{2,7\}$	$A \cup B$	$A \cup B$
$\{\mathbf{2},\mathbf{9}\}$	$\{c_i\} \cup a_{i-\frac{m}{2}\to i-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1}$	$\{c_i\} \cup a_{i-\frac{m}{2}\to i+\frac{m}{8}-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1}$
$\{\mathbf{5},\mathbf{7}\}$	V	V
$\{5,9\}$	C	C
$\{\mathbf{7},\mathbf{9}\}$	$\{c_i\} \cup A \cup B$	$\{c_i\} \cup A \cup B$
$\{1,5,2\}$	V	V
$\{1,5,7\}$	V	V
$\{1,5,9\}$	$C \cup a_{i\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1}$	$C \cup a_{i-\frac{m}{8}\to i+\frac{m}{2}-1} \cup b_{i-\frac{m}{8}\to i+\frac{m}{2}-1}$
$\{1,9,2\}$	$\{c_i\} \cup A \cup B$	$\{c_i\} \cup A \cup B$
$\{1,9,7\}$	$\{c_i\} \cup A \cup B$	$\{c_i\} \cup A \cup B$
$\{2,5,7\}$	V	V
$\{2,5,9\}$	$C \cup a_{i-\frac{m}{2}\to i-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1}$	$C \cup a_{i-\frac{m}{2}\to i+\frac{m}{8}-1} \cup b_{i-\frac{m}{2}\to i+\frac{m}{8}-1}$
$\{2,9,7\}$	$\{c_i\} \cup A \cup B$	$\{c_i\} \cup A \cup B$
$\{5,7,9\}$	V	V

That equality remains primitive in $\mathcal{L}(\mathit{full\text{-}disj}, \mathit{closed})$ already follows from our given proof in Section 4.1.3. In effect, the attentive reader may have noticed that we already cover full disjointness in that proof. In contrast, our proof of primitivity of disjointness in $\mathcal{L}(\mathit{eq}, \mathit{closed})$ does not extend to full equality. Nevertheless, we can reuse our proof of primitivity of *full* disjointness as follows. The graphs G and G' from Proposition 4.33 are indistinguishable in $\mathcal{L}(\mathit{full\text{-}eq}, \mathit{disj}, \mathit{closed})$. Let H and H' be the same graphs but with all directed edges reversed (i.e., the graphs illustrated in Figure 4.3). Then the same proof shows that H and H' are indistinguishable in $\mathcal{L}(\mathit{full\text{-}eq}, \mathit{closed})$. However, since G and G' are distinguishable by the inclusion statement $\exists p^{-}.\top \subseteq \neg \mathit{disj}(p^{-}, q^{-})$, also H and H' are distinguishable by the inclusion statement $\exists p.\top \subseteq \neg \mathit{disj}(p, q)$. Thus, the primitivity of disjointness in $\mathcal{L}(\mathit{full\text{-}eq}, \mathit{closed})$ is established.

4.4 Extension to stratified recursion

Until now, we could do without shape names. We do need them, however, for recursive shape schemas. Such schemas allow shapes to be defined using recursive rules, much as in Datalog and logic programming. The rules have a shape name in the head; in the body they have a shape that can refer to the same or other shape names.

Example 4.35. The following rule defines a shape, named s, recursively:

$$s \leftarrow \mathit{hasValue}(c) \lor (\mathit{eq}(p, q) \land \exists r.\mathit{hasShape}(s)).$$

A node x will satisfy s if there is a (possibly empty) path of r-edges from x to the constant c, so that all nodes along the path satisfy $\mathit{eq}(p, q)$ (for two property names p and q).

Rules and programs We need to make a few extensions to our formalism and the semantics.

- We assume an infinite supply S of *shape names.* Again for simplicity of notation only, we assume that S is disjoint from N and P.

- The syntax of shapes is extended so that *every shape name is a shape.*

- A vocabulary Σ is now a subset of $N \cup P \cup S$; an interpretation I now additionally assigns a subset $[\![s]\!]^I$ of Δ^I to every shape name s in Σ.

Noting the obvious parallels with the field of logic programming, we propose to use the following terminology from that field. A *rule* is of the form $s \leftarrow \varphi$, where s is a shape name and φ is a shape. A *program* is a finite set of rules. The shape names appearing as heads of rules in a program are called the *intensional* shape names of that program.

The following definitions of the semantics of programs are similar to definitions well-known for Datalog. A program is *semipositive* if for every intensional shape name s, and every shape φ in the body of some rule, s occurs only positively in φ. Let $\mathcal{P}$ be a semipositive program over vocabulary Σ, with set of intensional shape names D. An interpretation J over $\Sigma \cup D$ is called a *model* of $\mathcal{P}$ if for every rule $s \leftarrow \varphi$ of $\mathcal{P}$, the set $[\![\varphi]\!]^J$ is a subset of $[\![s]\!]^J$. Given any interpretation I over $\Sigma - D$, there exists a unique minimal interpretation J that expands I to $\Sigma \cup D$ such that J is a model of $\mathcal{P}$ (Indeed, J is the least fixpoint of the well-known immediate consequence operator, which is a monotone operator since $\mathcal{P}$ is semipositive [2]). We call J the result of applying $\mathcal{P}$ to I, and denote J by $\mathcal{P}(I)$.

Stratified programs are essentially sequences of semipositive programs. Formally, a program $\mathcal{P}$ is called *stratified* if it can be partitioned into parts $\mathcal{P}_1, \ldots, \mathcal{P}_n$ called *strata*, such that **(i)** the strata have pairwise disjoint sets of intensional shape names; **(ii)** each stratum is semipositive; and **(iii)** the strata are ordered in such a way that when a shape name s occurs in the body of a rule in some stratum, s is not intensional in any later stratum.

Let $\mathcal{P}$ be a stratified program with n strata $\mathcal{P}_1, \dots, \mathcal{P}_n$ and let again I be an interpretation over a vocabulary without the intensional shape names. We define $\mathcal{P}(I)$, the result of applying $\mathcal{P}$ to I, to be the interpretation J_n, where $J_0 := I$ and $J_{k+1} := \mathcal{P}_{k+1}(J_k)$ for $0 \leq k < n$.

Stratified shape schemas We are now ready to define a *stratified shape schema* again as a set of inclusions, but now paired with a stratified program. Formally, it is a pair $(\mathcal{P}, \mathcal{T})$, where:

- $\mathcal{P}$ is a program that is stratified, and where every shape name mentioned in the body of some rule is an intensional shape name in $\mathcal{P}$.
- $\mathcal{T}$ is a finite set of inclusion statements $\varphi_1 \subseteq \varphi_2$, where φ_1 and φ_2 mention only shape names that are intensional in $\mathcal{P}$.

Now we define a graph G to *conform* to $(\mathcal{P}, \mathcal{T})$ if $[\![\varphi_1]\!]^{\mathcal{P}(G)}$ is a subset of $[\![\varphi_2]\!]^{\mathcal{P}(G)}$, for every inclusion $\varphi_1 \subseteq \varphi_2$ in $\mathcal{T}$.

Remark 4.36. The nonrecursive notion of shape schema, defined in Chapter 2, corresponds to the special case where $\mathcal{P}$ is the empty program.

Extending Theorem 4.1 Theorem 4.1 extends to stratified shape schemas. Indeed, consider a stratified shape schema $(\mathcal{P}, \mathcal{T})$. Shapes not mentioning any shape names are referred to as *elementary shapes.* We observe that for every intensional shape name s and every graph H, there exists an elementary shape φ such that $[\![s]\!]^{\mathcal{P}(H)} = [\![\varphi]\!]^H$. Furthermore, φ uses the same constants, quantifiers, and path expressions as $\mathcal{P}$. For semipositive programs, this is shown using a fixpoint characterization of the minimal model; for stratified programs, this argument can then be applied repeatedly. The crux, however, is that graphs G and G' of Proposition 4.2 will have the same φ. Indeed, by that Proposition, the fixpoints of the different strata will be reached on G and on G' in the same stage. We effectively obtain an extension of Proposition 4.2, which establishes the theorem for features X other than *closed*.

Also for $X =$ *closed*, the reasoning, given after Lemma 4.17, extends in the same way to stratified shape schemas, since the graphs G and G' used there again yield exactly the same evaluation for all shapes that do not use *closed*.

Extending Theorem 4.20 Also Theorem 4.20 extends to stratified shape schemas. Thereto, Lemma 4.17 needs to be reproven in the presence of a stratified program $\mathcal{P}$ defining the intensional shape names. The extended Lemma 4.17 then states that $[\![\varphi]\!]^{\mathcal{P}(G)} = [\![\varphi]\!]^{\mathcal{P}(G')}$. The proof of Theorem 4.20 then goes through unchanged.

Extending Theorem 4.25 Also Theorem 4.25 extends to stratified shape schemas for the same reasons given above for Theorem 4.1.

5
Provenance

An important functionality expected of modern data management systems [1] is that they can provide *provenance* for the results they produce in response to queries or constraint checks. Intuitively, the provenance of a query result explains why the result was produced. Provenance typically takes the form of a subinstance, containing the data on which the produced result depends, or the data that is responsible for the result.

Provenance semantics have been proposed for a variety of data models and query languages, as surveyed by Glavic [47], even with many different proposals for the standard relational model and conjunctive queries. For SHACL, however, a provenance semantics has been lacking so far. The goal of this chapter is to fill this gap.

Provenance semantics is normally defined for query languages, not for constraint languages. Yet, any shape (constraint) φ can be naturally treated as the query that returns the set of nodes from the input graph that conform to φ. Following this idea, we will propose a provenance semantics for SHACL that returns, for any shape φ, any RDF graph G, and any focus node v from G that conforms to the shape, a certain subset of G. This subset, which we call the *neighborhood* of v in G with respect to φ, intuitively consists of the triples from G that contribute to v conforming to φ.

Example 5.1. Consider a publication graph (like the DBLP database) in RDF, where nodes represent papers, authors, and classes. We have :author-labeled edges from papers to their authors, and rdf:type-labeled edges from nodes to their class (e.g., paper, student, professor). Consider the constraint "the focus node has at least one author of type student", which can be written as the shape:

$$\mathsf{:WorkshopShape} \leftarrow \geq_1 \mathsf{:author}. \geq_1 \mathsf{rdf{:}type}.\mathit{hasValue}(\mathsf{:Student})$$

We will define the neighborhood of a conforming node v to consist of all triples (v :author x) from the graph where the graph also has the triple (x rdf:type Student), and that triple is also included in the neighborhood.

The above example involves a simple positive-existential constraint, but SHACL has quite powerful logical constructs, including negation, universal and counting quantifiers, path expressions, and primitives for equality and disjointness. This means that giving a nontrivial definition of neighborhood is challenging, if we want neighborhoods to satisfy an essential criterion known as *sufficiency* [47]. Simply put, a neighborhood N of a node v with respect to a shape φ is sufficient if v still conforms to φ when evaluated in the subgraph N. We will prove sufficiency for our provenance semantics for SHACL.

For conjunctive queries or positive-existential queries, sufficiency is easy to satisfy. For a language with the logical constructs mentioned above, however, we are the first to present a nontrivial provenance semantics for which sufficiency can be proved. We specify "nontrivial" here, as one can always define the neighborhood to be the entire graph and obtain sufficiency trivially. Indeed, the challenge is to keep only the relevant triples, without throwing out too much. Also, thanks to negation, we obtain both "why" and "why not" provenance [57]: if v does not conform to a shape φ, then its neighborhood for the shape $\neg\varphi$ provides the explanation.

Interestingly, neighborhoods suggest an opportunity to leverage shapes beyond conformance checking, and use them also to *retrieve* data. Specifically, given a shape φ and an input graph G, we can retrieve the subgraph of G formed by the union of all neighborhoods of all nodes in G that conform to φ. We refer to the result as the *shape fragment* of G with respect to φ. We will actually prove a stronger version of sufficiency, to the effect that a node v conforms to φ in G if and only if it conforms to φ in the shape fragment of G with respect to φ. In our work, we will duly generalize the notion of shape fragments to shape schemas, and also extend the sufficiency result to them.

Throughout this chapter, we use the complete formalization of SHACL as described in Section 2.4 with a slightly deviating definition of a shape schema. Here, a shape schema H is simply a collection of triples of the form (s, φ, τ) where $s \in S$ a shape name, φ a shape expression and τ the associated target declaration. If some named shapes do not have a target declaration, τ is $\bot$.

5.1 Data provenance for SHACL

In this section, we propose a provenance semantics for SHACL by defining the fundamental notion of the *neighborhood* of a node v for a shape φ in a graph G. The intuition is that this neighborhood consists of those triples in G that show that v conforms to φ; if v does not conform to φ, the neighborhood is set to be empty. We want a generic, tractable, deterministic definition that formalizes this intuition.

5.1.1 Neighborhoods

Before developing the definition formally, we discuss the salient features of our approach.

Negation Following the work by Grädel and Tannen on supporting where-provenance in the presence of negation [48], we assume shapes are in *negation normal form*, i.e., negation is only applied to atomic shapes. This is no restriction, since every shape can be put in negation normal form, preserving the overall syntactic structure, simply by pushing negations down. We push negation through conjunction and disjunction using De Morgan's laws. We push negation through quantifiers as follows:

$$\neg \geq_{n+1} E.\psi \equiv \leq_n E.\psi \quad \neg \leq_n E.\psi \equiv \geq_{n+1} E.\psi \quad \neg\forall E.\psi \equiv \geq_1 E.\neg\psi$$

The negation of $\geq_0 E.\psi$ is simply false.

Node tests We leave the neighborhood for *hasValue* and *test* shapes empty, as these involve no properties, i.e., no triples.

Closedness We also define the neighborhood for *closed*(P) to be empty, as this is a minimal subgraph in which the shape is indeed satisfied. A reasonable alternative approach would be to return all properties of the node, as "evidence" that these indeed involve only IRIs in P. Indeed, we will show in Section 5.1.4 that our definitions, while minimalistic, are taken such that they can be relaxed without sacrificing the sufficiency property.

Disjointness Still according to our minimal approach, the neighborhood for disjointness shapes is empty. Analogously, the same holds for *lessThan* and *uniqueLang* shapes.

Equality The neighborhood for a shape $eq(E, p)$ consists of the subgraph traced out by the E-paths and p-properties of the node under consideration, evidencing that the sets of end-nodes are indeed equal. Here, we can no longer afford to return the empty neighborhood, although equality would hold trivially there. Indeed, this would destroy the relaxation property promised above. For example, relaxing by adversely adding just one E-path and one p-property with distinct end-nodes, would no longer satisfy equality.

Nonclosure The neighborhood for a shape $\neg closed(P)$ consists of those triples from the node under consideration that involve properties outside P, as expected.

Nonequality For $\neg eq(E, p)$ we return the subgraph traced out by the E-paths from the node v under consideration that end in a node that is *not* a p-property of v, and vice versa. A similar approach is taken for nondisjointness and negated *lessThan* shapes.

Quantifiers The neighborhood for a shape $\forall E.\psi$ consists, as expected, of the subgraph traced out by all E-paths from the node under consideration to nodes x, plus the ψ-neighborhoods of these nodes x. For $\geq_n E.\psi$ we do something similar, but we take only those x that conform to ψ. Given the semantics of the $\geq_n$ quantifier, it seems tempting to instead just take a selection of n of such nodes x. However, we want a deterministic definition of neighborhood, so we take all x. Dually, for $\leq_n E.\psi$, we return the subgraph traced out by E-paths from the current node to nodes *not* conforming to ψ, plus their $\neg\psi$-neighborhoods.

5.1.2 Formal definition

Towards a formalization of the above ideas, we first make precise the intuitive notion of a path in an RDF graph, and of the subgraph traced out by a path. Paths are finite sequences of adjacent steps. Each step either moves forward from the subject to the object of a triple, or moves backward from the object to the subject. We make backward steps precise by introducing, for each property $p \in I$, its *reverse*, denoted by p^-. The set of reverse IRIs is denoted by I^-. We assume I and I^- are disjoint, and moreover, we also define $(p^-)^-$ to be p for every $p \in I$.

For any RDF triple $t = (s, p, o)$, the triple $t^- := (o, p^-, s)$ is called a *reverse triple*. As for IRIs, we define $(t^-)^-$ to be t. A *step* is an RDF triple (a forward step) or a reverse triple (a backward step). For any step $t = (x, r, y)$, we refer to x as the *tail*, denoted by $tail(t)$, and to y as the *head*, denoted by $head(t)$. A *path* is a nonempty finite sequence π of steps so that $head(t_1) = tail(t_2)$ for any two subsequent steps t_1 and t_2 in π. The *tail* of π is the tail of its first step; the *head* of π is the head of its last step. Any two paths π and π' where $head(\pi) = tail(\pi')$ can be concatenated; we denote this by $\pi \cdot \pi'$.

The *graph* traced out by a path π, denoted by $graph(\pi)$, is simply the set of RDF triples underlying the steps of the path. Thus, backward steps must be reversed. Formally,

$$graph(\pi) = \{t \mid t \text{ forward step in } \pi\} \cup \{t^- \mid t \text{ backward step in } \pi\}.$$

For a set Π of paths, we define $graph(\Pi) = \bigcup\{graph(\pi) \mid \pi \in \Pi\}$.

We are not interested in arbitrary sets of paths, but in the set of paths generated by a path expression E in an RDF graph G, denoted by $paths(E, G)$ and defined in a standard manner as follows.

- $paths(p, G) = \{(a, r, b) \in G \mid r = p\}$;

- $paths(E/E', G) = \{\pi \cdot \pi' \mid \pi \in paths(E, G) \;\&\; \pi' \in paths(E', G) \;\&\; head(\pi) = tail(\pi')\}$;
- $paths(E \cup E', G) = paths(E, G) \cup paths(E', G)$;
- $paths(E?, G) = paths(E, G)$;
- $paths(E^*, G) = \bigcup_{i=1}^{\infty} paths(E^i, G)$; and
- $paths(E^-, G) = \{\pi^- \mid \pi \in paths(E, G)\}$.

Here, E^i abbreviates $E/\cdots/E$ (i times), and $\pi^- = t_l^-, \dots, t_1^-$ for $\pi = t_1, \dots, t_l$. Note that $paths(p, G)$ is a set of length-one paths.

In order to link E-paths to the evaluation of shapes below, we introduce some more notation, for any two nodes a and b:

$$paths(E, G, a, b) := \{\pi \in paths(E, G) \mid tail(\pi) = a \;\&\; head(\pi) = b\}$$

Note that $graph(\pi)$, for every $\pi \in paths(E, G)$, is a subgraph of G. This will ensure that neighborhoods and shape fragments are always subgraphs of the original graph. Moreover, the following observation ensures that path expressions will have the same semantics in the neighborhood as in the original graph:

Proposition 5.2. *Let $F = graph(paths(E, G, a, b))$. Then $(a, b) \in [\![E]\!]^G \Leftrightarrow (a, b) \in [\![E]\!]^F$.*

Proof. That $[\![E]\!]^F \subseteq [\![E]\!]^G$ is immediate from $F \subseteq G$ and the monotonicity of path expressions. For the reverse inclusion, we proceed by induction on the structure of E. The base case, where E is a property p, is immediate from the definitions. The inductive cases where E is one of $E_1 \cup E_2$, E_1^-, or E_1/E_2, are clear. Two cases remain:

- Let E be of the form $E_1?$. If $a = b$, then $(a, b) \in [\![E]\!]^F$ by definition. Otherwise, (a, b) must be in $[\![E_1]\!]^G$. Therefore, by induction, $(a, b) \in [\![E_1]\!]^F \subseteq [\![E]\!]^F$.
- Let E be of the form E_1^*. If $a = b$, then $(a, b) \in [\![E]\!]^F$ by definition. Otherwise, there exist $i \geq 1$ nodes $c_0, \dots, c_i$ such that $a = c_0$ and $b = c_i$, and $(c_j, c_{j+1}) \in [\![E_1]\!]^G$ for $0 \leq j < i$. By induction, each $(c_j, c_{j+1}) \in [\![E_1]\!]^F$, whence (a, b) belongs to the transitive closure of $[\![E_1]\!]^F$ as desired. □

Note that $paths(E, G)$ may be infinite, due to the use of Kleene star in E and cycles in G. However $graph(paths(E, G))$ is always finite, because G is finite.

We are now ready to define neighborhoods in the context of an arbitrary but fixed schema H. To avoid clutter we will omit H from the notation.

Definition 5.3. Let v be a node, G be a graph, and φ be a shape. We define the *φ-neighborhood of v in G*, denoted by $B(v, G, \varphi)$, as the empty RDF graph whenever v does not conform to φ in G. When v does conform, the definition is given in Table 5.1.

In the table, as already discussed above, by pushing negations down, we can and do assume that φ is put in *negation normal form*, meaning that negation is only applied to atomic shapes. (Atomic shapes are those from the grammar in Section 2.4 that do not refer back to φ.)

Example 5.4. Consider the "happy at work" shape

$$\neg disj(\text{:friend}, \text{:colleague})$$

The neighborhood of a conforming node v consists of the union of all pairs of triples (v :friend x) and (v :colleague x) for each common x that exists in the data graph.

Table 5.1: Neighborhood $B(v, G, \varphi)$ in the context of a schema H, when $G, v \models \varphi$ and φ is in negation normal form. In particular, in rules 2 and 6, we assume that $\neg \mathit{def}(s, H)$ and $\neg\psi$ are put in negation normal form. In the omitted cases, and when $G, v \not\models \varphi$, the neighborhood is defined to be empty.

φ	$B(v, G, \varphi)$
$\mathit{hasShape}(s)$	$B(v, G, \mathit{def}(s, H))$
$\neg\mathit{hasShape}(s)$	$B(v, G, \neg\mathit{def}(s, H))$
$\varphi_1 \wedge \varphi_2$	$B(v, G, \varphi_1) \cup B(v, G, \varphi_2)$
$\varphi_1 \vee \varphi_2$	$B(v, G, \varphi_1) \cup B(v, G, \varphi_2)$
$\geq_n E.\psi$	$\bigcup\{\mathit{graph}(\mathit{paths}(E, G, v, x)) \cup B(x, G, \psi) \mid (v, x) \in \llbracket E \rrbracket^G \ \\& \ G, x \models \psi\}$
$\leq_n E.\psi$	$\bigcup\{\mathit{graph}(\mathit{paths}(E, G, v, x)) \cup B(x, G, \neg\psi) \mid (v, x) \in \llbracket E \rrbracket^G \ \\& \ G, x \models \neg\psi\}$
$\forall E.\psi$	$\bigcup\{\mathit{graph}(\mathit{paths}(E, G, v, x)) \cup B(x, G, \psi) \mid (v, x) \in \llbracket E \rrbracket^G\}$
$\mathit{eq}(E, p)$	$\bigcup\{\mathit{graph}(\mathit{paths}(E \cup p, G, v, x)) \mid (v, x) \in \llbracket E \cup p \rrbracket^G\}$
$\mathit{eq}(\mathit{id}, p)$	$\{(v, p, v)\}$
$\neg\mathit{eq}(E, p)$	$\bigcup\{\mathit{graph}(\mathit{paths}(E, G, v, x)) \mid (v, x) \in \llbracket E \rrbracket^G \ \\& \ (v, p, x) \notin G\} \cup$ $\{(v, p, x) \in G \mid (v, x) \notin \llbracket E \rrbracket^G\}$
$\neg\mathit{eq}(\mathit{id}, p)$	$\{(v, p, x) \in G \mid x \neq v\}$
$\neg\mathit{disj}(E, p)$	$\bigcup\{\mathit{graph}(\mathit{paths}(E, G, v, x)) \cup \{(v, p, x)\} \mid (v, x) \in \llbracket E \rrbracket^G \ \\& \ (v, p, x) \in G\}$
$\neg\mathit{disj}(\mathit{id}, p)$	$\{(v, p, v)\}$
$\neg\mathit{lessThan}(E, p)$	$\bigcup\{\mathit{graph}(\mathit{paths}(E, G, v, x)) \cup \{(v, p, y)\} \mid (v, x) \in \llbracket E \rrbracket^G \ \\&$ $(v, p, y) \in G \ \\& \ x \not< y\}$
$\neg\mathit{lessThanEq}(E, p)$	$\bigcup\{\mathit{graph}(\mathit{paths}(E, G, v, x)) \cup \{(v, p, y)\} \mid (v, x) \in \llbracket E \rrbracket^G \ \\&$ $(v, p, y) \in G \ \\& \ x \not\leq y\}$
$\neg\mathit{uniqueLang}(E)$	$\bigcup\{\mathit{graph}(\mathit{paths}(E, G, v, x)) \mid (v, x) \in \llbracket E \rrbracket^G \ \\& \ \exists y \neq x : (v, y) \in \llbracket E \rrbracket^G \ \\& \ y \sim x\}$
$\neg\mathit{closed}(P)$	$\{(v, p, x) \in G \mid p \notin P\}$

5.1.3 Algorithms for neighborhoods

Table 5.1 defines neighborhoods by set-theoretic expressions which are constructive, comparable to safe relational calculus formulas in the relational model [95]. As such, these expressions immediately yield a naive algorithm for computing neighborhoods.

Consider, for example, the computation of $B(v, G, \geq_n E.\psi)$. We proceed as follows, following the set-theoretic expression provided. Run through all nodes x for which there is an E-path from v to x. Algorithms for such regular path queries are well understood [11] and are supported by SPARQL query processors. For each such x, test whether $G, x \models \psi$. This test is according to the semantics of shapes defined in Table 2.1, which is again constructive and algorithmic. Now for each x passing the test, recursively compute $B(x, G, \psi)$, and also compute $\mathit{graph}(\mathit{paths}(E, G, v, x))$. Collect the results for all x, and return their union.

For another example, the computation of $B(v, G, \neg\mathit{eq}(E, p))$ proceeds as follows. Again run through all nodes x for which there is an E-path from v to x. For each x we test if (v, p, x) is in G; if it is not, we compute $\mathit{graph}(\mathit{paths}(E, G, v, x))$. We collect the resulting triples for all x in a temporary result set T_1. Secondly, we run through all nodes x for which the triple $t = (v, p, x)$ is in G. For each x we test if x is reachable from v by an E-path; if it is not, add t to the temporary result set T_2. We finally return the union $T_1 \cup T_2$.

All cases from Table 5.1 likewise can be given an algorithmic reading, so together they provide a (naive) algorithm for computing neighborhoods.

Computing $\mathit{graph}(\mathit{paths}(E, G, v, x))$**.** A key ingredient in the neighborhood algorithm is the computation of the subgraph $\mathit{graph}(\mathit{paths}(E, G, v, x))$. For simple path expressions E which are just a property p or an inverse property p^-, these are simply the singletons $\{(v, p, x)\}$ and $\{(x, p, v)\}$, respectively. For more complex path expressions E, however, it is not obvious how $\mathit{graph}(\mathit{paths}(E, G, v, x))$ can be computed. We will actually show this later in Lemma 5.13, by effectively reducing the problem to the computation of a SPARQL query.

Complexity In Section 5.2.2 we will see more generally that, in fact, the entire neighborhood can be computed by a single SPARQL query. Since SPARQL (without the need for regular expressions with counting, and using standard regular path semantics) has polynomial-time data complexity [74], we obtain polynomial-time complexity of neighborhood computation.

5.1.4 The sufficiency property

We can prove that neighborhoods indeed provide us with an adequate provenance semantics for shapes. Specifically, we want to show that the neighborhood $B(v, G, \varphi)$ is sufficient in the sense of providing provenance for the conformance of v to φ in G. Thinking of a shape as a unary query, returning all nodes that conform to it, the following theorem states exactly the "sufficiency property" that has been articulated in the theory of data provenance [47].

Theorem 5.5 (Sufficiency). *If $G, v \models \varphi$ then also $G', v \models \varphi$ for any RDF graph G' such that $B(v, G, \varphi) \subseteq G' \subseteq G$.*

Proof. For any shape φ, we consider its expansion with relation to the schema H, obtained by repeatedly replacing subshapes of the form *hasShape*(s) by *def*(s, H), until we obtain an equivalent shape that no longer contains any subshapes of the form *hasShape*(s).

Proof by induction on the height of the expansion of φ in negation normal form, where the height of negated atomic shapes is defined to be zero. When φ is $\top$, *test*(t), or *hasValue*(c), and v conforms to φ in G, then v clearly also conforms to φ in G', as the conformance of the node is independent of the graph. We start by considering the remaining base cases.

Let φ be of the form *eq*(E, p). We must show that $[\![E]\!]^{G'}(v) = [\![p]\!]^{G'}(v)$. For the containment from left to right, let $x \in [\![E]\!]^{G'}(v)$. Since E is monotone, $x \in [\![E]\!]^{G}(v)$. Since $G, v \models \varphi$, $x \in [\![p]\!]^{G}(v)$. Let $F = \mathit{graph}(\mathit{paths}(p, G, v, x))$. By Proposition 5.2, $x \in [\![p]\!]^{F}(v)$. By definition of φ-neighborhood, $F \subseteq B$, and we know $B \subseteq G'$. Therefore, because path expressions are monotone, we also have $x \in [\![p]\!]^{G'}(v)$ as desired. The containment from right to left is analogous.

Let φ be of the form *eq*(id, p). We must show that $[\![\mathit{id}]\!]^{G'}(v) = [\![p]\!]^{G'}(v)$, or equivalently we must show that $\{v\} = [\![p]\!]^{G'}(v)$. We know that $G, v \models \varphi$, therefore $[\![p]\!]^{G}(v) = \{v\}$. Now we only need to show that $(v, p, v) \in G'$ as $G' \subseteq G$ (and therefore G' does not contain more p-edges than G). Because by definition of neighborhood $B = \{(v, p, v)\}$, and because $B \subseteq G'$, the claim follows.

Let φ be of the form *disj*(E, p). Let $x \in [\![E]\!]^{G'}(v)$. Since E is monotone, $x \in [\![E]\!]^{G}(v)$. Since $G, v \models \varphi$, $x \notin [\![p]\!]^{G}(v)$. Therefore, as p is monotone, $x \notin [\![p]\!]^{G'}(v)$. The case where $x \in [\![p]\!]^{G'}(v)$ is analogous.

Let φ be of the form *disj*(id, p). We must show that $(v, p, v) \notin G'$. Because $G, v \models \varphi$, we know that $(v, p, v) \notin G$. As $G' \subseteq G$, the claim follows.

Let φ be of the form *lessThan*(E, p). Let $x \in [\![E]\!]^{G'}(v)$. Let $(v, p, y) \in G'$. We must show that $x < y$. Since E is monotone, $x \in [\![E]\!]^{G}(v)$ and since $G' \subseteq G$, $(v, p, y) \in G$. As $G, v \models \varphi$, we know that $x < y$ in G and thus also in G'.

The case where φ is the form *lessThanEq*(E, p) is analogous to the previous case.

Let φ be of the form *uniqueLang*(E). Let $x \in [\![E]\!]^{G'}(v)$. Let $y \in [\![E]\!]^{G'}(v)$ such that $y \neq x$. As E is monotone, $x \in [\![E]\!]^{G}(v)$ and $y \in [\![E]\!]^{G}(v)$. As $G, v \models \varphi$, we know $y \nsim x$ in G and thus also in G'.

Let φ be of the form *closed*(P). Let $(v, p, x) \in G'$. Then, $(v, p, x) \in G$. Therefore, as $G, v \models \varphi$, $p \in P$ as desired.

Let φ be of the form $\neg$*eq*(E, p). Since $G, v \models \varphi$, there are two cases. First, there exists a node $x \in [\![E]\!]^{G}(v)$ such that $x \notin [\![p]\!]^{G}(v)$. Let $F = \mathit{graph}(\mathit{paths}(E, G, v, x))$. By Proposition 5.2, $x \in [\![E]\!]^{F}(v)$. By definition of φ-neighborhood $F \subseteq B$, and we know

$B \subseteq G'$. Therefore, since E is monotone, $x \in [\![E]\!]^{G'}(v)$. Next, since $(v,p,x) \notin G$, we know $(v,p,x) \notin G'$. Thus, $[\![E]\!]^{G}(v) \neq [\![p]\!]^{G}(v)$ as desired. For the other case, there exists a node x such that $(v,p,x) \in G$ and $x \notin [\![E]\!]^{G}(v)$. By definition of φ-neighborhood, $(v,p,x) \in B \subseteq G'$. However, because E is monotone $x \notin [\![E]\!]^{G'}(v)$. Therefore $[\![p]\!]^{G}(v) \neq [\![E]\!]^{G}(v)$ as desired.

Let φ be of the form $\neg eq(id,p)$. Since $G,v \models \varphi$, there are two cases. First, $(v,p,v) \notin G$. We know $G' \subseteq G$, therefore if $(v,p,v) \notin G$, then $(v,p,v) \notin G'$ as desired. Second, $(v,p,v) \in G$ and there exists a node x such that $(v,p,x) \in G$ and $x \neq v$. From the definition of neighborhood, we know that this also holds for B and therefore also in G' as $B \subseteq G'$.

Let φ be of the form $\neg disj(E,p)$. Since $G,c \models \varphi$, we know that there exists a node $x \in [\![E]\!]^{G}(v)$ such that $(v,p,x) \in G$. Let $F = graph(paths(E,G,v,x))$. By Proposition 5.2, $x \in [\![E]\!]^{F}(v)$. By definition of φ-neighborhood $F \subseteq B$, and we know $B \subseteq G'$. Then, since E is monotone, $x \in [\![E]\!]^{G'}(v)$. Next, by definition of φ-neighborhood, also $(v,p,x) \in B \subseteq G'$. Thus, $x \in [\![E]\!]^{G'}(v) \cap [\![p]\!]^{G'}(v)$ as desired.

Let φ be of the form $\neg disj(id,p)$. We need to show that $(v,p,v) \in G'$. By definition of neighborhood, $(v,p,v) \in B$. As $B \subseteq G'$, $(v,p,v) \in G'$ as desired.

Let φ be of the form $\neg lessThan(E,p)$. Since $G,v \models \varphi$, there exists a node $x \in [\![E]\!]^{G}(v)$ and a node $y \in [\![p]\!]^{G}(v)$ with $x \not< y$. If we can show that $x \in [\![E]\!]^{G'}(v)$ and $x \in [\![p]\!]^{G'}(v)$, it will follow that $G',v \models \varphi$ as desired. Let $F = graph(paths(E,G,v,x))$. By Proposition 5.2, $x \in [\![E]\!]^{F}(v)$. By definition of φ-neighborhood, $F \subseteq B$, and we know $B \subseteq G'$. Then, since E is monotone, $x \in [\![E]\!]^{G'}(v)$. Next, by definition of φ-neighborhood, $(v,p,x) \in B$. Since $B \subseteq G'$, also $x \in [\![p]\!]^{G'}(v)$ as desired.

The case where φ is the form $\neg lessThanEq(E,p)$ is analogous to the previous case.

Let φ be of the form $\neg uniqueLang(E)$. Since $G,v \models \varphi$, there exists $x_1 \neq x_2 \in [\![E]\!]^{G}(v)$ such that $x_1 \sim x_2$. As in the previous case, we must show that x_1 and x_2 are in $[\![E]\!]^{G'}(v)$. By Proposition 5.2, for both $i = 1,2$, we have $x_i \in [\![E]\!]^{F_i}(v)$ with $F_i = graph(paths(E,G,v,x_i))$. By definition of φ-neighborhood $F_i \subseteq B \subseteq G'$. Therefore $x_i \in [\![E]\!]^{G'}(v)$ as desired.

Let φ be of the form $\neg closed(P)$. As $G,v \models \varphi$, there exists a property $p \notin P$ and a node x such that $(v,p,x) \in G$. By definition $(v,p,x) \in B(v,G,\varphi) \subseteq G'$. Hence, $G',v \models \varphi$ as desired.

We proceed with the inductive cases. Let φ be of the form $\varphi_1 \wedge \varphi_2$. By induction, we know v conforms to φ_1 in G' and conforms to φ_2 in G'. Therefore, v conforms to $\varphi_1 \wedge \varphi_2$ in G'.

Let φ be of the form $\varphi_1 \vee \varphi_2$. We know v conforms to at least one of φ_i for $i \in \{1,2\}$ in G. Assume w.l.o.g. that v conforms to φ_1 in G. Then, our claim follows by induction.

Let φ be of the form $\geq_n E.\psi$. Here, and in the following cases, we denote $B(v,G,\varphi)$ by B. As $G,v \models \varphi$, we know there are at least n nodes $x_1,\dots,x_n$ in G such that $x_i \in [\![E]\!]^{G}(v)$ and $G,x_i \models \psi$ for all $1 \leq i \leq n$. Let $F = graph(paths(E,G,v,x))$. By Proposition 5.2, $x_i \in [\![E]\!]^{F}(v)$. By definition of φ-neighborhood $F \subseteq B$, and we know $B \subseteq G'$. Therefore, because E is monotone, $x_i \in [\![E]\!]^{G'}(v)$. Furthermore, since $B(x_i,G,\psi) \subseteq B \subseteq G'$, by induction, $G',x_i \models \psi$ as desired.

Let φ be of the form $\leq_n E.\psi$. First we show that every $x \in [\![E]\!]^{G'}(v)$ that conforms to ψ in G', must also conform to ψ in G.

Proof by contradiction. Suppose there exists a node $x \in [\![E]\!]^{G'}(v)$ that conforms to ψ in G', but conforms to $\neg\psi$ in G. By definition of φ-neighborhood, $B(x,G,\neg\psi) \subseteq B$, and we know $B \subseteq G'$. Therefore, by induction, x conforms to $\neg\psi$ in G', which is a contradiction.

Because of the claim above, the number of nodes reachable from v through E that satisfy ψ in G' must be smaller or equal to the number of nodes reachable from v through E that satisfy ψ in G. Because we know $G,v \models \varphi$, the lemma follows.

Let φ be of the form $\forall E.\psi$. For all nodes x such that $x \in [\![E]\!]^{G'}(v)$, as E is monotone, $x \in [\![E]\!]^{G}(v)$. As $G,v \models \varphi$, $G,x \models \psi$. By definition of φ-neighborhood, $B(x,G,\psi) \subseteq B$. We know $B \subseteq G'$. Thus, by induction, $G', x \models \psi$ as desired. □

Note that the Sufficiency property is stated not just for the neighborhood, but more strongly for all subgraphs G' that encompass the neighborhood. This stronger statement serves both a technical and a practical purpose. The technical purpose is that it is needed to deduce our results on shape fragments (cf. the next Subsection). The practical advantage is that it allows some leeway for provenance engines. Indeed, even if the engine, for reasons of efficiency or ease of implementation, return *larger* neighborhoods than the ones we strictly define, Sufficiency will continue to hold.

Example 5.6. Let us consider the a schema about workshop papers and authors. We require that each paper must have at least one author, but can have at most one author who is *not* of type student. These two constraints are captured by a schema H with two shape definitions. One has the shape expression $\geq_1 \mathsf{author}.\top$, and the other has the shape expression

$$\leq_1 \mathsf{author}.\neg \geq_1 \mathsf{type}.\mathit{hasValue}(\mathsf{student}),$$

which in negation normal form becomes

$$\leq_1 \mathsf{author}. \leq_0 \mathsf{type}.\mathit{hasValue}(\mathsf{student}).$$

Both shape definitions have target $\geq_1 \mathsf{type}.\mathit{hasValue}(\mathsf{paper})$. We denote the two shape expressions by φ_1 and φ_2, and the target by τ.

Consider the simple graph G consisting of a single paper, say $\mathsf{p1}$. This paper has two authors: Anne, who is a professor, and Bob, who is a student. Formally, G consists of the five triples $(\mathsf{p1}, \mathsf{type}, \mathsf{paper})$, $(\mathsf{p1}, \mathsf{auth}, \mathsf{Anne})$, $(\mathsf{p1}, \mathsf{auth}, \mathsf{Bob})$, $(\mathsf{Anne}, \mathsf{type}, \mathsf{prof})$ and $(\mathsf{Bob}, \mathsf{type}, \mathsf{student})$.

Let us consider the neighborhood of $\mathsf{p1}$ for the shape $\varphi_1 \wedge \tau$. This neighborhood consists of the three triples $(\mathsf{p1}, \mathsf{type}, \mathsf{paper})$, $(\mathsf{p1}, \mathsf{auth}, \mathsf{Anne})$ and $(\mathsf{p1}, \mathsf{auth}, \mathsf{Bob})$. On the other hand, the neighborhood of $\mathsf{p1}$ for $\varphi_2 \wedge \tau$ consists of the three triples $(\mathsf{p1}, \mathsf{type}, \mathsf{paper})$, $(\mathsf{p1}, \mathsf{auth}, \mathsf{Bob})$ and $(\mathsf{Bob}, \mathsf{type}, \mathsf{student})$.

Note that the triple $(\mathsf{Bob}, \mathsf{type}, \mathsf{student})$ is essential in the neighborhood for $\varphi_2 \wedge \tau$; omitting it would break Sufficiency. On the other hand, we are free to add the triple $(\mathsf{Anne}, \mathsf{type}, \mathsf{prof})$ to any of the neighborhoods without breaking Sufficiency.

Finally, note that we could add to G various other triples unrelated to the shapes φ_1, φ_2 and τ. The neighborhoods would omit all this information, as desired.

We conclude this section with a number of remarks.

Remark 5.7. A natural question is whether $B(v, G, \varphi)$, as we have defined it, is *minimal* while still being sufficient in the sense of Theorem 5.5. Our discussion on quantifiers in Section 5.1.1 already indicated non-minimality. Assume, for example, that φ is "the focus node must have an a-property" (say, an address), expressed as $\geq_1 a.\top$. In a graph G with two triples (v, a, x) and (v, a, y), node v has two addresses. Any of the two triples in itself would be sufficient as a neighborhood of v for φ. Choosing between the two addresses x and y, however, leads to a nondeterministic behavior.

Remark 5.8. In Theorem 5.1.1, what can we say if v does not conform to φ in G? In this case, v conforms to $\neg\varphi$ in G. Hence, the provenance for the non-conformance will be provided by $B(v, G, \neg\varphi)$. This point was first made by Köhler, Ludäscher and Zinn [57], who, however, do not define neighborhood subgraphs and do not prove any Sufficiency property.

Remark 5.9. The neighborhood of a node, for whatever shape, is always a subset of its connected component in the graph. Thus, Sufficiency implies that only the connected component (indeed, only the neighborhood!) is needed to check conformance of a node.

5.2 Shape fragments

In this section we define and illustrate the idea of shape fragments as a novel mechanism to retrieve subgraphs.

The *shape fragment* of an RDF graph G, for a finite set S of shapes, is the subgraph of G formed by the neighborhoods of all nodes in G for the shapes in S. Formally:

$$\mathit{Frag}(G, S) = \bigcup \{B(v, G, \varphi) \mid v \in N \ \& \ \varphi \in S\}.$$

Here, v ranges over the universe N of all nodes, but since neighborhoods are always subgraphs of G, it is equivalent to let v range over all subjects and objects of triples in G. So, to compute $\mathit{Frag}(G, S)$, we run over v, retrieve the neighborhoods for each v independently, and collect and return the set of resulting triples.

The shapes in S can be interpreted as arbitrary "request shapes". An interesting special case, however, is when S is derived from a shape schema H. Formally, we define the shape fragment of G for H as $\mathit{Frag}(G, H) := \mathit{Frag}(G, S)$, where $S = \{\varphi \wedge \tau \mid \exists s : (s, \varphi, \tau) \in H\}$. Thus, the shape fragment for a schema requests the conjunction of each shape in the schema with its associated target.

In order to state our main results concerning these two types of shape fragments, we need to revisit the definition of schema. Recall that a schema is a set of shape definitions, where a shape definition is of the form (s, φ, τ). Until now, we allowed both the shape expression φ and the target τ to be arbitrary shapes. In real SHACL, however, only shapes of the following specific forms can be used as targets:

- $\mathit{hasValue}(c)$ (node targets);
- $\geq_1 p/r^*.\mathit{hasValue}(c)$ (class-based targets: p and r stand for type and subclass from the RDF Schema vocabulary [79], and c is the class name);
- $\geq_1 p.\top$ (subjects-of targets); and
- $\geq_1 p^-.\top$ (objects-of targets).

For our purposes, however, what counts is that real SHACL targets τ are *monotone*, in the sense that if $G, v \models \tau$ and $G \subseteq G'$, then also $G', v \models \tau$.

We establish:

Theorem 5.10 (Conformance). *Assume that schema H has monotone targets, and assume RDF graph G conforms to H. Then $\mathit{Frag}(G, H)$ also conforms to H.*

Proof. The proof is a straightforward application of Theorem 5.5. Let $F = \mathit{Frag}(G, H)$; we must show that F conforms to H. Thereto, consider a shape definition $(s, \varphi, \tau) \in H$, and let v be a node such that $F, v \models \tau$. Since $F \subseteq G$ and τ is monotone, also $G, v \models \tau$, whence $G, v \models \varphi$ since G conforms to H. Since by definition, F contains $B(v, G, \varphi)$, Sufficiency yields $F, v \models \varphi$ as desired. □

Moreover, Sufficiency carries over to shape fragments defined by arbitrary request shapes as follows:

Corollary 5.11. *Let G be an RDF graph, let S be a finite set of shapes, let φ be a shape in S, and let v be a node. If $G, v \models \varphi$, then also $\mathit{Frag}(G, S), v \models \varphi$.*

Example 5.12. For monotone shapes, the converse of Corollary 5.11 clearly holds as well. In general, however, the converse does not always hold. For example, consider the shape $\varphi = \leq_0 p.\top$ ("the node has no property p"), and the graph $G = \{(a, p, b)\}$. Then the fragment $\mathit{Frag}(G, \{\varphi\})$ is empty, so a trivially conforms to φ in the fragment. However, a clearly does not conform to φ in G.

5.2.1 Applicability of shape fragments

In order to assess the practical applicability of shape fragments, we simulated a range of SPARQL queries by shape fragments. Queries were taken from the SPARQL benchmarks BSBM [15] and WatDiv [5]. Unlike a shape fragment, a SPARQL select-query does not return a subgraph but a set of variable bindings. SPARQL construct-queries do return RDF graphs directly, but not necessarily subgraphs. Hence, we followed the methodology of modifying SPARQL select-queries to construct-queries that return all *images* of the pattern specified in the where-clause.

For tree-shaped basic graph patterns, with given IRIs in the predicate position of triple patterns, we can always simulate the corresponding subgraph query by a shape fragment. Indeed, a typical query from the benchmarks retrieves nodes with some specified properties, some properties of these properties, and so on. For example, a slightly simplified WatDiv query, modified into a subgraph query, would be the following. (To avoid clutter, we forgo the rules of standard IRI syntax.)

```
CONSTRUCT WHERE {
  ?v0 caption ?v1 .
  ?v0 hasReview ?v2 .
  ?v2 title ?v3 .
  ?v2 reviewer ?v6 .
  ?v7 actor ?v6 }
```

(Here, CONSTRUCT WHERE is the SPARQL notation for returning all images of a basic graph pattern.) We can express the above query as the fragment for the following request shape:

$$\geq_1 \mathsf{caption}.\top \wedge \geq_1 \mathsf{hasReview}.(\geq_1 \mathsf{title}.\top \wedge \geq_1 \mathsf{reviewer}. \geq_1 \mathsf{actor}^-.\top)$$

Of course, patterns can involve various SPARQL operators, going beyond basic graph patterns. Filter conditions on property values can be expressed as node tests in shapes; optional matching can be expressed using $\geq_0$ quantifiers. For example, consider a simplified version of the pattern of a typical BSBM query:

```
?v text ?t . FILTER langMatches(lang(?t),"EN") OPTIONAL { ?v rating ?r }
```

The images of this pattern can be retrieved using the shape

$$\geq_1 \mathsf{title}.\mathit{test}(\mathsf{lang} = \text{“EN”}) \wedge \geq_0 \mathsf{rating}.\top.$$

Interestingly, the BSBM workload includes a pattern involving a combination of optional matching and a negated bound-condition to express absence of a certain property (a well-known trick [7, 9]). Simplified, this pattern looks as follows:

```
?prod label ?lab .
?prod feature 870
OPTIONAL { ?prod feature 59 . ?prod label ?var }
FILTER (!bound(?var))
```

The images of this pattern can be retrieved using the shape

$$\geq_1 \mathsf{label}.\top \wedge \geq_1 \mathsf{feature}.\mathit{hasValue}(870) \wedge \leq_0 \mathsf{feature}.\mathit{hasValue}(59).$$

A total of 39 out of 46 benchmark queries, modified to return subgraphs, could be simulated by shape fragments in this manner. The remaining seven queries involved features not supported by SHACL, notably, variables in the property position, or arithmetic.

5.2.2 Translation to SPARQL

Our first approach to computing neighborhoods is by translation into SPARQL, the recommended query language for RDF graphs [50]. SPARQL select-queries return sets of *solution mappings*, which are maps μ from finite sets of variables to N. Variables are marked using question marks. Different mappings in the result may have different domains [10, 73].

Neighborhoods in a graph G are unions of subgraphs of the form $\mathit{graph}(\mathit{paths}(E, G, a, b))$, for path expressions E mentioned in the shapes, and selected nodes a and b. Hence, the following lemma is important. For any RDF graph G, we denote by $N(G)$ the set of all subjects and objects of triples in G.

Lemma 5.13. *For every path expression E, there exists a SPARQL select-query $Q_E(?t, ?s, ?p, ?o, ?h)$ such that for every RDF graph G:*

1. *The binary relation $\{(\mu(?t), \mu(?h)) \mid \mu \in Q_E(G)\}$ equals $[\![E]\!]^G$, restricted to $N(G)$.*
2. *For all $a, b \in N(G)$, the RDF graph*

$$\{(\mu(?s), \mu(?p), \mu(?o)) \mid \mu \in Q_E(G) \ \&\ (\mu(?t), \mu(?h)) = (a, b) \\ \&\ \mu \textit{ is defined on } ?s,\ ?p \textit{ and } ?o\}$$

equals $\mathit{graph}(\mathit{paths}(E, G, a, b))$.

Proof. The difficulty is that we do not merely have to test if $(a, b) \in [\![E]\!]^G$, which can readily be expressed using SPARQL property paths, but that we actually have to return $\mathit{graph}(\mathit{paths}(E, G, a, b))$. We construct Q_E by induction on the structure of E. We list Q_E in each of the cases of the syntax of path expressions. In the base case, when E is a property name p:

```
SELECT (?s AS ?t) ?s ($p$ AS ?p) ?o (?o AS ?h) WHERE { ?s $p$ ?o }
```

When E is of the form E_1^-, we obtain Q_{E_1} by induction, and construct Q_E as follows:

```
SELECT (?h AS ?t) ?s ?p ?o (?t AS ?h) WHERE { $Q_{E_1}$ }
```

When E is of the form $E_1?$:

```
SELECT ?t ?s ?p ?o ?h WHERE {
  { $Q_{E_1}$ } UNION
  { SELECT (?v AS ?t) (?v AS ?h)
    WHERE { { ?v ?_p1 ?_o1 } UNION { ?_s2 ?_p2 ?v } } } }
```

When E is of the form $E_1 \cup E_2$:

```
SELECT ?t ?s ?p ?o ?h WHERE { { $Q_{E_1}$ } UNION { $Q_{E_2}$ } }
```

When E is of the form E_1/E_2:

```
SELECT ?t ?s ?p ?o ?h WHERE {
  { { SELECT ?t ?s ?p ?o (?h AS ?h1) WHERE { $Q_{E_1}$ } } .
    { SELECT (?t AS ?h1) ?h WHERE { ?t $E_2$ ?h } } }
  UNION {
  { SELECT ?t (?h AS ?h1) WHERE { ?t $E_1$ ?h } } .
  { SELECT (?t AS ?h1) ?s ?p ?o ?h WHERE { $Q_{E_2}$ } } } }
```

Finally, when E is of the form E_1^*:

```
SELECT ?t ?s ?p ?o ?h WHERE {
   { ?t $E_1^*$ ?x1 . ?x2 $E_1^*$ ?h .
     { SELECT (?t AS ?x1) ?s ?p ?o (?h AS ?x2) WHERE { $Q_{E_1}$ } } }
   UNION {
     SELECT (?v AS ?h) (?v AS ?t)
     WHERE { { ?v ?_p1 ?_o1 } UNION { ?_s2 ?_p2 ?v } } } }
```
□

The following example illustrates the lemma, but using a more readable query than the one that would be literally generated by the above proof.

Example 5.14. For IRIs a, b, q and r, the following SPARQL query, applied to any graph G, returns $\mathit{graph}(\mathit{paths}((q/r)^*, G, a, b))$:

```
SELECT ?s ?p ?o
WHERE { $a$ $(q/r)$* ?t . ?h $(q/r)$* $b$ . {
     { SELECT ?t (?t AS ?s) ($q$ AS ?p) ?o ?h
       WHERE { ?t $q$ ?o . ?o $r$ ?h } }
     UNION
     { SELECT ?t ?s ($r$ AS ?p) (?h AS ?o) ?h
       WHERE { ?t $q$ ?s . ?s $r$ ?h } } } }
```
□

Using Lemma 5.13, and expressing the definitions from Table 5.1 in SPARQL, we obtain that neighborhoods can be uniformly computed in SPARQL as follows.

Proposition 5.15. *For every shape φ, there exists a SPARQL select-query $Q_\varphi(?v, ?s, ?p, ?o)$ such that for every RDF graph G,*

$$\{(\mu(?v), \mu(?s), \mu(?p), \mu(?o)) \mid \mu \in Q_\varphi(G)\} = \{(v, s, p, o) \in N^4 \mid (s, p, o) \in B(v, G, \varphi)\}$$

Moreover, the size of Q_φ is linear in the size of φ.

Proof. As always we work in the context of a schema H. We assume φ is put in negation normal form and proceed by induction as in the proof of the Sufficiency Theorem.

Note that Q_φ should not merely check conformance of nodes to shapes, but actually must return the neighborhoods. Indeed, that conformance checking in itself is possible in SPARQL (for nonrecursive shapes) is well known; it was even considered for recursive shapes [29]. Hence, in the constructions below, we use an auxiliary SPARQL query $CQ_\varphi(?v)$ (C for conformance) which returns, on every RDF graph G, the set of nodes $v \in N(G)$ such that $G, v \models \varphi$. We now describe Q_φ for all the cases in the following. We start by noting that many shapes have an empty neighborhood, concretely, the cases where φ is of one of the forms: $\top$, $\mathit{hasValue}(c)$, $\mathit{test}(t)$, $\mathit{closed}(P)$, $\mathit{disj}(E, p)$, $\mathit{disj}(\mathit{id}, p)$, $\mathit{lessThan}(E, p)$, $\mathit{lessThanEq}(E, p)$, $\mathit{uniqueLang}(E)$, $\neg\top$, $\neg\mathit{hasValue}(c)$, and $\neg\mathit{test}(t)$ have an empty neighborhood, i.e., the corresponding SPARQL query is the empty query. Furthermore, when φ is of the form $\mathit{hasShape}(s)$, the query is given by $Q_{\mathit{def}(s,H)}$. Analogously, when φ is of the form $\neg\mathit{hasShape}(s)$, the query is given by $Q_{\neg\mathit{def}(s,H)}$. We proceed with listing all other cases.

When φ is of the form $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$, we obtain Q_{φ_1} and Q_{φ_2} by induction, and construct Q_φ as follows:

```
SELECT ?v ?s ?p ?o
WHERE { { $CQ_\varphi$ } . { $Q_{\varphi_1}$ } UNION { $Q_{\varphi_2}$ }}
```

When φ is of the form $\geq_n E.\varphi_1$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
    { { SELECT (?v AS ?t) WHERE { $CQ_{\varphi}$ } } .
      { $Q_E$ } .
      { SELECT (?v AS ?h) WHERE { $CQ_{\varphi_1}$ } }
    } UNION
    { { SELECT (?v AS ?t) WHERE { $CQ_{\varphi}$ } } .
      ?t $E$ ?h .
      { SELECT (?v AS ?h) ?s ?p ?o
        WHERE { { $Q_{\varphi_1}$ } . { $CQ_{\varphi_1}$ }}}}}
```

When φ is of the form $\leq_n E.\varphi_1$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
    { { SELECT (?v AS ?t) WHERE { $CQ_{\varphi}$ } } .
      { $Q_E$ } .
      { SELECT (?v AS ?h) WHERE { $CQ_{\neg\varphi_1}$ } }
    } UNION
    { { SELECT (?v AS ?t) WHERE { $CQ_{\varphi}$ } } .
      ?t $E$ ?h .
      { SELECT (?v AS ?h) ?s ?p ?o
        WHERE { { $Q_{\neg\varphi_1}$ } . { $CQ_{\neg\varphi_1}$ }}}}}
```

When φ is of the form $\forall E.\varphi_1$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
    { { SELECT (?v AS ?t) WHERE { $CQ_{\varphi}$ } } .
      { $Q_E$ }
    } UNION
    { { SELECT (?v AS ?t) WHERE { $CQ_{\varphi}$ } } .
      ?t $E$ ?h .
      { SELECT (?v AS ?h) ?s ?p ?o
        WHERE { $Q_{\varphi_1}$ }}}}
```

When φ is of the form $eq(id, p)$:

```
SELECT ?v (?s AS ?v) ($p$ AS ?p) (?v AS ?o)
WHERE { { $CQ_{\varphi}$ } . ?v $p$ ?v }
```

When φ is of the form $\neg closed(P)$):

```
SELECT ?v (?v AS ?s) ?p ?o
WHERE { { $CQ_{\varphi}$ } . ?v ?p ?o FILTER (?p NOT IN $P$) }
```

When φ is of the form $\neg eq(E, p)$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
    { SELECT (?v AS ?t) WHERE { $CQ_{\varphi}$ } } .
    { { { $Q_E$ } MINUS { ?t $p$ ?h } }
      UNION
      { { $Q_p$ } MINUS { ?t $E$ ?h } } } }
```

When φ is of the form $\neg eq(id, p)$:

```
SELECT ?v (?v AS ?s) ($p$ AS ?p) (?v AS ?o)
WHERE { { $CQ_\varphi$ } . ?v $p$ ?o FILTER (?o != ?v) }
```

When φ is of the form $\neg \mathit{disj}(E, p)$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
    { SELECT (?v AS ?t) WHERE { $CQ_\varphi$ } } .
    { { { $Q_E$ } . { ?t $p$ ?h } }
      UNION
      { { $Q_p$ } . { ?t $E$ ?h } } } }
```

When φ is of the form $\neg \mathit{disj}(\mathit{id}, p)$:

```
SELECT ?v (?v AS ?s) ($p$ AS ?p) (?v AS ?o)
WHERE { { $CQ_\varphi$ } . ?v $p$ ?v }
```

When φ is of the form $\neg \mathit{lessThan}(E, p)$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
    { SELECT (?v AS ?t) WHERE { $CQ_\varphi$ } } .
    { { { $Q_E$ } . { ?t $p$ ?h2 } FILTER (! ?h ¡ ?h2) }
      UNION
      { { $Q_p$ } . { ?t $E$ ?h2 } FILTER (! ?h2 ¡ ?h) } } }
```

When φ is of the form $\neg \mathit{lessThanEq}(E, p)$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
    { SELECT (?v AS ?t) WHERE { $CQ_\varphi$ } } .
    { { { $Q_E$ } . { ?t $p$ ?h2 } FILTER (! ?h ¡= ?h2) }
      UNION
      { { $Q_p$ } . { ?t $E$ ?h2 } FILTER (! ?h2 ¡= ?h) } } }
```

And finally, when φ is of the form $\neg \mathit{uniqueLang}(E)$:

```
SELECT (?t AS ?v) ?s ?p ?o
WHERE {
    { SELECT (?v AS ?t) WHERE { $CQ_\varphi$ } } .
    { $Q_E$ } . { ?t $E$ ?h2 }
    FILTER (?h != ?h2 && lang(?h) = lang(?h2)) }
```

□

The linear-size claim can indeed be verified by inspecting the construction in the proof of Lemma 5.13 and the construction of Q_φ.

Remark 5.16. The above result should not be confused with the known result [29, Proposition 3] that SPARQL can compute the set of nodes that *conform* to a given shape. Our result states that also the neighborhoods can be computed. □

Since shape fragments are unions of neighborhoods, we also obtain:

Corollary 5.17. *For every finite set S of shapes, there exists a SPARQL select-query $Q_S(?s, ?p, ?o)$ such that for every RDF graph G,*

$$\{(\mu(?s), \mu(?p), \mu(?o)) \mid \mu \in Q_S(G)\} = \mathit{Frag}(G, S).$$

Example 5.18. For IRIs p, q and c, consider the request shape $\forall p. \geq_1 q.\mathit{hasValue}(c)$ (e.g., "all my friends like ping-pong", with p, q and c playing the role of friend, like, and ping-pong, respectively). The corresponding shape fragment is retrieved by the following SPARQL query:

```
SELECT ?s ?p ?o WHERE {
{ SELECT ?v WHERE
  { ?v p ?x MINUS { ?v p ?y OPTIONAL { ?y q c . ?v p ?z }
                               FILTER (!bound(?z)) } } } .
{ { SELECT (?v AS ?s) (p AS ?p) (?x as ?o)
    WHERE { ?v p ?x . ?x q c } }
  UNION
  { SELECT (?x AS ?s) (q AS ?p) (c as ?o)
    WHERE { ?v p ?x . ?x q c } } } }
```

The first subselect retrieves nodes $?v$ conforming to the shape; the UNION of the next two subselects then retrieves the neighborhoods. □

The above example illustrates that query expressions for shapes can quickly become quite complex, even for just retrieving the nodes that conform to a shape. Shapes involving equality constraints require nested not-exists subqueries in SPARQL, and would benefit from specific operators for set joins, e.g., [51,63]. Shapes of the form $\leq_5 p.\top$ requires grouping the p-properties and applying a condition count ≤ 5, plus a union with an outer join to retrieve the nodes without any p-property. Such shapes would benefit from specific operators for group join [38, 64]. Query optimization for queries derived from SHACL is an important topic for further research.

One may wonder about the converse to Corollary 5.17: is every SPARQL select-query expressible as a shape fragment? This does not hold, if only because shape fragments always consist of triples from the input graph, while select-queries can return arbitrary variable bindings. However, also more fundamentally, SHACL is strictly weaker than SPARQL; we give two representative examples.

4-clique Let $p \in I$. There does not exist a shape φ such that, on any RDF graph G, the nodes that conform to φ are exactly the nodes belonging to a 4-clique of p-triples in G. We can show that if 4-clique would be expressible by a shape, then the corresponding 4-clique query about a binary relation P would be expressible in 3-variable counting infinitary logic $C^3_{\infty\omega}$. The latter is known not to be the case, however [66]. (Infinitary logic is needed here to express path expressions, and counting is needed for the $\geq_n$ quantifier, since we have only 3 variables.)

Majority Let $p, q \in I$. There does not exist a shape φ such that, on any RDF graph G, the nodes that conform to φ are exactly the nodes v such that $\sharp\{x \mid (v,p,x) \in G\} \geq \sharp\{x \mid (v,q,x) \in G\}$ (think of departments with at least as many employees as projects). We can show that if Majority would be expressible by a shape, then the classical Majority query about two unary relations P and Q would be expressible in first-order logic. Again, the latter is not the case [58]. (Infinitary logic is not needed here, since for this query, we can restrict to a class of structures where all paths have length one.)

5.2.3 Adapting a validation engine

We have also investigated computing neighborhoods by adjusting a SHACL validator to return the validated RDF terms and their neighborhood, instead of a validation report.

A SHACL validation engine checks whether a given RDF graph conforms to a given schema, and produces a validation report detailing possible violations. A validation engine needs to inspect the neighborhoods of nodes anyway. Hence, it requires only reasonably lightweight adaptations to produce, in addition to the validation report, also the nodes and their neighborhoods that validate the shapes graph, without introducing significant overheads for tracing out and returning these neighborhoods, compared to doing validation alone. Our hypothesis is that the resulting overhead will not be prohibitive.

To test this hypothesis, we extended the open-source, free-license engine pySHACL [76]. This is a main-memory engine and it achieves high coverage for core SHACL [44]. Written in Python, we found it easy to make local changes to the code [71]; starting out with 4501 lines of code, 482 lines were changed, added or deleted.

Our current implementation covers most of SHACL core, with the exception of complex path expressions, i.e., only simple path expressions are supported. The algorithm that is implemented is then essentially the naive algorithm described in Section 5.1.3.

Our software, called *pySHACL-fragments*, is available open-source [77].

5.2.4 Experiments

We validated our approach by (i) measuring the overhead of neighborhood extraction, compared to mere validation, using pySHACL-fragments; and (ii) testing the viability of computing neighborhoods by translation to SPARQL. We perform our experiments in the context of computing shape fragments. Indeed, shape fragments offer a natural test case as they require the neighborhoods of all nodes to be retrieved.

Extraction overhead

To evaluate the viability of computing neighborhoods by adapting a validation engine, we measured the overhead of extracting neighborhoods using our system pySHACL-fragments, compared to producing the corresponding validation report using pySHACL alone. Performance evaluation of SHACL engines is not our goal here; see Schaffenrath et al. [83] for this. Yet, we reuse the 57 shapes from their performance benchmark. These shapes are expressed over a 30-million triple dataset known as the “Tyrolean Knowledge Graph”. Notably, however, Schaffenrath et al. managed to run their comparative study on a 1-million slice of the knowledge graph only, as common SHACL validation engines are still in their infancy and not very efficient.

For our experiment, instead, we generated a 1.5-million triple induced subgraph of the knowledge graph as follows. We sampled 50 000 individual nodes uniformly at random, and then retrieved all triples involving these individuals as subjects or objects. By sampling a larger number of 100K, 150K and 200K nodes, we similarly obtained subgraphs of (approximately) 2.5, 3.5, and 4.5 million triples.

We used a 12 core AMD EPYC 2.595GHz processor with 16GB DDR4 RAM and 400GB SSD. We executed each of the shapes three times, both on pySHACL and on pySHACL-fragments. Timers were placed around the `validator.run()` function, so *data loading and shape parsing time is not included.* The average overhead turns out to be well below 10%; if we restrict attention to the shapes where validation on the 1.5M graph takes longer than a second, the average overhead grows to 15.6%. Figure 5.1 shows that the overhead may vary somewhat going to larger graphs, but stays constant on average. There are some outliers where the overhead fluctuates more wildly, but these happen to be associated with low (below second) runtimes.

The shapes where the overhead is highest are those with existential shapes and many target nodes with large neighborhoods. For some property p and some condition ψ, an existential shape requires that the target node must have at least one p-edge to a node x satisfying ψ (expressed as $\geq_1 p.\psi$). Here, a validator merely needs to check for each target

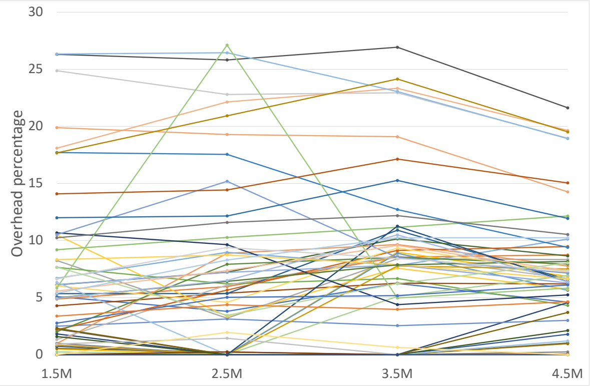


Figure 5.1: Overhead (percent increase in time to do provenance extraction, over mere validation of a shape) shown for 57 shapes over four graph sizes. Each line represents a shape.

node v that at least one such x exists, while provenance computation must also retrieve all the satisfying triples (v, p, x).

Computing neighborhoods in SPARQL

Instead of modifying an existing SHACL engine, one may compute provenance using SPARQL queries, as presented in Section 5.2.2. Shapes give rise to complex SPARQL queries which pose quite a challenge to SPARQL query processors. It is outside the scope of the present study to do a performance study of SPARQL query processors; our goal rather is to obtain an indication of the practical feasibility of computing neighborhoods in SPARQL.

Initial work by Corman et al. has reported satisfying results on doing *validation* for nonrecursive schemas by a single, complex SPARQL query [29]. The question we want to answer is whether we can observe a similar situation when computing neighborhoods, where the queries become even more complex.

We have obtained a mixed picture. We used the main-memory SPARQL engine Apache Jena ARQ. Implementing Corollary 5.17 by following the constructive proof of Proposition 5.15, we translated the shape fragment queries for the benchmark shapes from the previous Section 5.2.4 into large SPARQL queries. The generated expressions can be hundreds of lines long, as our translation procedure is not yet optimized to generate “efficient” SPARQL expressions. However, we then reduced the shapes by substituting $\top$ for node tests, and simplifying the resulting expressions. This reduction preserves the graph-navigational nature of the queries.

After the reduction, 13 out of 57 shapes produced SPARQL queries that ARQ could execute. The other queries were still too long and did not terminate or went out of memory. Figure 5.2 shows the runtimes on the same test data and the same machine as the overhead experiment; one shape is omitted from the Figure as it does not retrieve any triples at all. Reported timings are averages over three runs.

Finally, to test the extraction of paths in SPARQL, we used the DBLP database [32], and computed the shape fragment for shape $\geq_1 a^-/a/a^-/a/a^-/a.\mathit{hasValue}(\mathsf{MYV})$, where a stands for the property dblp:authoredBy, and MYV stands for the DBLP IRI for Moshe Y. Vardi. This extracts not only all authors at co-author distance three or less from this famous computer scientist, but, crucially, also all a-triples on all the relevant paths. The generated SPARQL query is similar to the query from Example 5.14.

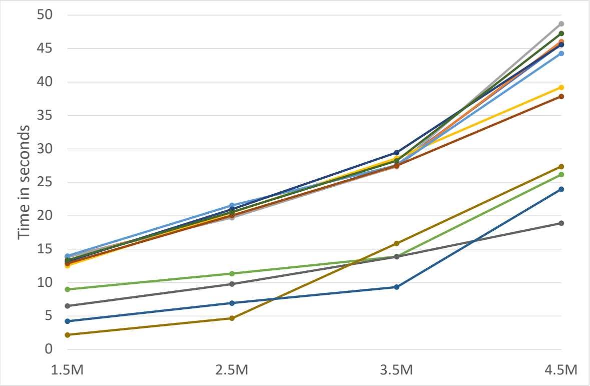


Figure 5.2: Execution times of provenance computation for 12 shapes by SPARQL queries, over four graph sizes. Each line represents a shape.

We ran this heavy analytical query on the two secondary-memory engines Apache Jena ARQ on TDB2 store, and GraphDB. The execution times over increasing slices of DBLP, going backwards in time from 2021 until 2010, are comparable between the two engines (see Figure 5.3). Vardi is a prolific and central author and co-author; just from 2016 until 2021, almost 7% of all DBLP authors are at distance three or less, or almost 145 943 authors. The resulting shape fragment contains almost 3% of all dblpl:authoredBy triples, or 219 085 unique triples. We see that retrieving neighborhoods can be a computationally intensive task for which new methods may be needed.

For the Vardi experiment we used a 2x 8core Intel Xeon E5-2650 v2 processor with 48GB DDR3 RAM and a 250GB hard disk.

Discussion

From these experiments, we conclude that computing neighborhoods is viable, but can be computationally intensive. Indeed, provenance for SHACL serves as an interesting new challenge and testbed both for SHACL validators (suitably adapted to retrieve neighborhoods) and SPARQL engines. Advances on either front will also benefit SHACL provenance performance. Interestingly, recent approaches to SHACL validation [29, 41] consider decomposing the task into multiple small SPARQL queries, as opposed to translating to a single large query.

5.3 Related work

Shapes may be viewed as queries on RDF graphs, returning the nodes that conform to the shape. This observation allows us to compare neighborhoods for shapes, with provenance semantics for queries proposed in the literature [27, 47].

A seminal work in the area of data provenance is that on *lineage* by Cui, Widom and Wiener [31]. Like neighborhoods, the lineage of a tuple returned by a query on a database D is a subdatabase of D. Lineage was defined for queries expressed in the relational algebra. In principle, we can express shapes in relational algebra. So, instead of defining our own notion of neighborhood, should we have simply used lineage instead? The answer is no; the following example shows that Sufficiency would fail.

Example 5.19. Recalling Example 5.6, consider a relational database schema with three relation schemes Paper(P), Author(P, A), and Student(A), and the query Q returning all

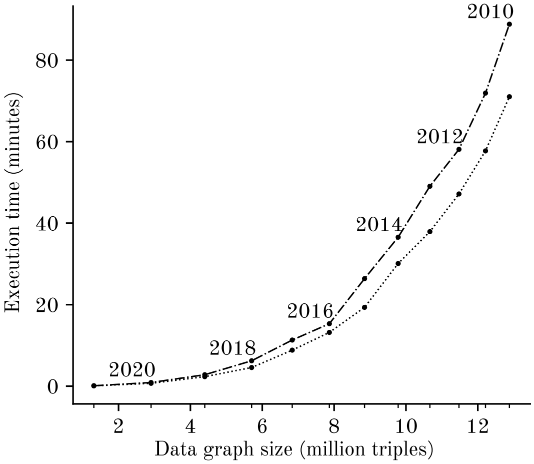


Figure 5.3: Jena ARQ store-based SPARQL execution time (dotted) and store-based GraphDB execution time (dashed-dotted) for the Vardi-distance-3 shape fragment.

papers with at least one author but without non-student authors. Consider the database D given by

$$\begin{aligned}&D(\text{Paper}) = \{\mathsf{p1}\};\\&D(\text{Author}) = \{(\mathsf{p1}, \mathsf{Bob})\};\\&D(\text{Student}) = \{\mathsf{Bob}\}.\end{aligned}$$

Note that $\mathsf{p1}$ is returned by Q on D. A relational algebra expression for Q is $E = \text{Paper} \bowtie (\pi_P(\text{Author}) - V)$ with $V = \pi_P(\text{Author} - (\text{Author} \bowtie \text{Student}))$. Since V is empty on D, the lineage of $\mathsf{p1}$ for E in D is the database D' where

$$D'(\text{Paper}) = \{\mathsf{p1}\};\ D'(\text{Author}) = \{(\mathsf{p1}, \mathsf{Bob})\};\ D'(\text{Student}) = \emptyset.$$

However, $\mathsf{p1}$ is no longer returned by E on D. □

An alternative approach to lineage is *why-provenance* [24] which is non-deterministic in that it reflects that there may be several “explanations” for why a tuple is returned by a query (for example, queries involving existential quantification). Accordingly, why-provenance does not yield a single neighborhood (called witness), but a set of them. While logical, this approach is at odds with our aim of providing a *deterministic* retrieval mechanism through shapes. Of course, one could take the union of all witnesses, but this runs into similar problems as illustrated in the above example. Indeed, why-provenance was not developed for queries involving negation or universal quantification.

A recent approach to provenance for negation is that by Grädel and Tannen [48,91] based on the successful framework of provenance semirings [49]. There, provenance is produced in the form of provenance polynomials which give a compact representation of the several possible proof trees showing that the tuple satisfies the query. Thus, like why-provenance, this approach is inherently non-deterministic. Still, we were influenced by Grädel and Tannen’s use of negation normal form, which we have followed in this work.

5.3.1 Triple pattern fragments

Shape fragments return subgraphs: they retrieve a subset of the triples of an input graph. A popular subgraph-returning mechanism is that of *triple pattern fragments* (TPF [102]). A TPF may indeed be viewed as a query that, on an input graph G, returns the subset of G consisting of all images of some fixed triple pattern in G.

While the logic of shapes is, in general, much richer than simple triple patterns, it turns out that not all TPFs are actually expressible by shape fragments.

For example, TPFs of the form $(?x, p, ?y)$, $(c, p, ?x)$, $(?x, p, c)$, or (c, p, d), for IRIs p, c, and d, are easily expressed as shape fragments using request shapes $\geq_1 p.\top$, $\geq_1 p^-.\mathit{hasValue}(c)$, $\geq_1 p.\mathit{hasValue}(c)$, or $\mathit{hasValue}(c) \wedge \geq_1 p.\mathit{hasValue}(d)$, respectively.

The TPF $(?x, p, ?x)$, asking for all p-self-loops in the graph, corresponds to the shape fragment for $\neg \mathit{disj}(\mathit{id}, p)$.

Furthermore, the TPFs $(?x, ?y, ?z)$ (requesting a full download) and $(c, ?y, ?z)$ are expressible using the request shapes $\neg\mathit{closed}(\emptyset)$ and $\mathit{hasValue}(c) \wedge \neg\mathit{closed}(\emptyset)$. Here, the need to use a "trick" via negation of closedness constraints exposes a weakness of shapes: properties are not treated on equal footing as subjects and objects. Indeed, other TPFs involving variable properties, such as $(?x, ?y, c)$, $(?x, ?y, ?x)$, or $(c, ?x, d)$, are not expressible as shape fragments.

The above discussion can be summarized as follows.

Proposition 5.20. *The TPFs expressible as a shape fragment (uniformly over all input graphs) are precisely the TPFs of the following forms:*

1. $(?x, p, ?y)$;
2. $(?x, p, c)$;
3. $(c, p, ?x)$;
4. (c, p, d);
5. $(?x, p, ?x)$;
6. $(?x, ?y, ?z)$;
7. $(c, ?y, ?z)$.

It remains to show that all other forms of TPF are not expressible as shape fragments. Since, for any finite set S of shapes, we can form the disjunction $\bigvee S$ of all shapes in S, and $\mathit{Frag}(G, S) = \mathit{Frag}(G, \{\bigvee S\})$ for any graph G, it suffices to consider single shapes φ instead of finite sets of shapes. We abbreviate $\mathit{Frag}(G, \{\varphi\})$ to $\mathit{Frag}(G, \varphi)$.

Formally, let $Q = (u, v, w)$ be a triple pattern, i.e., u, v and w are variables or elements of N. Let V be the set of variables from $\{u, v, w\}$ to N. A solution mapping is a function $\mu : V \to N$. For any node a, we agree that $\mu(a) = a$. Then the TPF query Q maps any input graph G to the subset

$$Q(G) = \{(\mu(u), \mu(v), \mu(w)) \mid \mu : V \to N \ \&\ (\mu(u), \mu(v), \mu(w)) \in G\}.$$

We now say that a shape φ *expresses* a TPF query Q if $\mathit{Frag}(G, \varphi) = Q(G)$ for every graph G.

We begin by showing:

Lemma 5.21. *Let G be an RDF graph and let φ be a shape. Assume $\mathit{Frag}(G, \varphi)$ contains a triple (s, p, o) where p is not mentioned in φ. Then $\mathit{Frag}(G, \varphi)$ contains all triples in G of the form (s, p', o'), where p' is not mentioned in φ.*

Proof. Since shape fragments are unions of neighborhoods, it suffices to verify the statement for an arbitrary neighborhood $B(v, G, \varphi)$. This is done by induction on the structure of the negation normal form of φ. In almost all cases of Table 5.1, triples from $B(v, G, \varphi)$ come from E-paths, with E mentioned in φ; from $B(v, G, \psi)$, with ψ a subshape of φ or the negation thereof; or involve a property p clearly mentioned in φ. Triples of the first kind never have a property not mentioned in φ, and triples of the second kind satisfy the statement by induction.

The only remaining case is $\neg \mathit{closed}(P)$. Assume (v, p, x) is in the neighborhood, and let $(v, p', x') \in G$ be a triple such that p' is not mentioned in φ. Then certainly $p' \notin P$, so (v, p', x') also belongs to the neighborhoods, as desired. □

Using the above Lemma, we give:

Proof of Proposition 5.20. Consider the TPF $Q = (?x, ?x, ?y)$ and assume there exists a shape φ such that $Q(G) = \mathit{Frag}(G, \varphi)$ for all G. Consider $G = \{(a, a, b), (a, c, b)\}$, where a and c are not mentioned in φ. We have $(a, a, b) \in Q(G)$ so $(a, a, b) \in \mathit{Frag}(G, \varphi)$. Then by Lemma 5.21, also $(a, c, b) \in \mathit{Frag}(G, \varphi)$. However, $(a, c, b) \notin P(G)$, so we arrive at a contradiction, and φ cannot exist.

Similar reasoning can be used for all other forms of TPF not covered by the proposition. Below we give the table of these TPFs Q, where c and d are arbitrary IRIs, possibly equal, and $?x$ and $?y$ are distinct variables. The right column lists the counterexample graph G showing that $Q(G) \neq \mathit{Frag}(G, \varphi)$. Importantly, the property (a or b) of the triples in G is always chosen so that it is not mentioned in φ, and moreover, a, b and e are distinct and also distinct from c and d.

Q	G
$(?x, ?y, ?x)$	$\{(a, b, a), (a, b, c)\}$
$(?x, ?y, ?y)$	$\{(a, b, b), (a, b, c)\}$
$(?x, ?x, ?x)$	$\{(a, a, a), (a, a, b)\}$
$(?x, ?y, c)$	$\{(a, b, c), (a, b, d)\}$
$(?x, ?x, c)$	$\{(a, a, c), (a, a, d)\}$
$(?x, ?y, ?y)$	$\{(a, b, b), (a, b, c)\}$
$(c, ?x, ?x)$	$\{(c, a, a), (c, a, b)\}$
$(c, ?x, d)$	$\{(c, a, d), (c, a, e)\}$

□

Remark 5.22. SHACL does not allow *negated properties* in path expressions, while these are supported in SPARQL property paths. Extending SHACL with negated properties would readily allow the expression of *all* TPFs as shape fragments. For example, the TPF $(?x, ?y, c)$, for IRI c, would become expressible by requesting the shape

$$\geq_1 p.\mathit{hasValue}(c) \vee \geq_1 !p.\mathit{hasValue}(c),$$

with p an arbitrary IRI. Here, the negated property $!p$ matches any property different from p.

5.3.2 Knowledge graph subsets

Recently, the idea of defining subgraphs (or fragments as we call them) using shapes was independently proposed by Labra Gayo [59]. An important difference with our SHACL-based approach is that his approach is based on ShEx, the other shape language besides SHACL that is popular in practice [23,45]. Shapes in ShEx are quite different from those in SHACL, being based on bag-regular expressions over the bag of properties of the focus node. As a

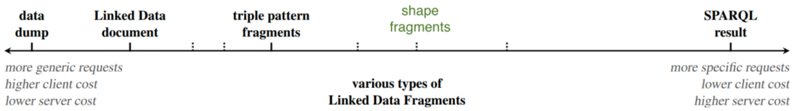


Figure 5.4: Positioning shape fragments in the LDF Framework (adapted from [102]). This diagram is not to be interpreted as a comparison in expressive power.

result, the technical developments of our work and Labra Gayo's are quite different. Still, the intuitive and natural idea of forming a subgraph by collecting all triples encountered during conformance checking, is clearly the same in both approaches. This idea, which Labra Gayo calls "slurping", is implemented in our pyshacl-fragments implementation, as well as a "slurp" option in the shex.js implementation of ShEx [87]. Labra Gayo also gives a formal definition of ShEx + slurp, extending the formal definition of ShEx [23].

In our work we make several additional contributions compared to the development by Labra Gayo:

- We make the connection to database provenance.
- We consider the important special case of shape fragments based on schemas with targets.
- We support path expressions directly, which in ShEx need to be expressed through recursion.
- We support negation, universal quantification, and other non-monotone quantifiers and shapes, such as $\leq_n$, equality, disjointness, lessThan.
- We establish formal correctness properties (Sufficiency and Conformance Theorems).
- We investigate the translation of shape fragments into SPARQL. On the other hand, Labra Gayo discusses Pregel-based implementations of his query mechanism.

5.3.3 Path-returning queries on graph databases

Our definition of neighborhood of a node v for a shape involving a path expression E returns E-paths from v to relevant nodes x (see Table 5.1). Notably, these paths are returned as a subgraph, using the *graph* constructor applied to a set of paths. Thus, shape fragments are loosely related to path-returning queries on graph databases, introduced as a theoretical concept by Barceló et al. [13] and found in the languages Cypher [43] and G-CORE [60].

However, to our knowledge, a mechanism to return a set of paths in the form of a subgraph is not yet implemented by these languages. We have showed in Section 5.2.2 that, at least in principle, this is actually possible in any standard query language supporting path expressions, such as SPARQL. Barceló et al. consider a richer output structure whereby an infinite set of paths (or even set of tuples of paths) resulting from an extended regular path query can be finitely and losslessly represented. In contrast, our *graph* constructor is lossy in that two different sets S_1 and S_2 of paths may have $\mathit{graph}(S_1) = \mathit{graph}(S_2)$. However, our Sufficiency property shows that our representation is sufficient for the purpose of validating shapes.

6 Conclusions

6.1 Formalization

In this thesis, starting from the formalizations of SHACL in the literature [6, 30], we have investigated the formalization of SHACL, establishing its nature as a (description) logic. This connection was also observed in previous work [3, 61, 69, 70], however, none of the discussed works, took the explicit viewpoint that a data graph represents a standard first-order interpretation or that SHACL validation is model checking. We took this viewpoint and in doing so formalized precisely how SHACL relates to the field of description logics. Later, this viewpoint was also adopted by Ortiz [65] and used by Seifer et al. [85], indicating that it is indeed natural. There are (at least) three reasons why this formalization is important. First, it establishes a bridge between two communities, thereby allowing to exploit the many years of research in Description Logics also for studying SHACL. Second, our formalization of SHACL clearly separates two orthogonal concerns:

1. Which information does a data graph represent? This is handled in the translation of a graph into its natural interpretation.

2. What is the semantics of language constructs? This is handled purely in the well-studied logical setting.

Third, as we showed above, our formalization corresponds closer to actual SHACL than existing formalizations, respects well-known laws (such as De Morgan's) and avoids issues with nodes not occurring in the graph requiring special treatment. We then extended this basic formalization such that it corresponds exactly to real SHACL. The construction in Section 2.4.1 can be seen as a formal definition of real SHACL.

Using this foundation, we can accurately study different aspects of SHACL like recursion, expressiveness and provenance. Furthermore, the logical syntax itself could be the basis of an alternative SHACL syntax to be used in practice. It highlights the fundamental features of SHACL without, for example, hiding the (universal, existential, and counting) quantification as much as the current SHACL syntax does, which may increase understanding, and avoid unintended bugs in shape graphs.

6.2 Recursion

Corman, Reutter and Savkovic [30] started with the work on recursive semantics and even already defined a three-valued operator. Hence, the only work left to obtain a rich family of semantics, was observing that this operator is indeed an approximator and applying AFT. As such, we believe our approach establishes strong and formal foundations for the study of recursive SHACL. Indeed, AFT does not just dictate how the semantics are to be defined, but immediately provides guarantees such as stratification and predicate introduction results that can be instrumental when developing concrete validators for recursive SHACL: we immediately obtain results about which transformations can safely be applied to our theories. Furthermore, we compare the semantics dictated by AFT with the existing literature. We do not directly propose any semantics for SHACL. It is important to realize though, that in case one wants to view a shapes as (inductive) definitions, it has been argued repeatedly that the Well-founded semantics correctly formalizes the notion of inductive definitions [37]. The Well-founded semantics was later also directly defined for SHACL, indicating interest in its adoption [28].

Nevertheless, whatever semantics for SHACL will be adopted by the W3C, it should be informed by what is actually desired by users and should be easy to understand. A sensible suggestion could be to only define recursion for SHACL shapes graphs where recursion through negation can only occur in a stratified manner, i.e., Stratified Negation as discussed in Section 1.3. This keeps the language simple, and many natural recursive shapes can be defined this way. Furthermore, Stable model- and Well-founded semantics agree on the semantics for stratified shapes graphs. So implementations can still choose to adopt these alternative semantics, while still adhering to the specification.

6.3 Expressiveness

We have established the primitivity of the non-standard logical features of SHACL. However, an obvious open question is whether our results extend further to nonstratified programs, under well-founded or stable model semantics. Notably, Corman et al. [30] have already suggested that disjointness is redundant in a setting of recursive shape schemas with non-stratified negation. Their expression is not correct, however [82].[1]

There is clear interest in the expressiveness of SHACL. There have been efforts to formalize Wikidata constraints in SHACL [40] where the authors argue certain constraints are not expressible in SHACL. The study of expressiveness also concerns practitioners who propose extensions for SHACL, like the DASH constraint components [56]. A natural question to ask is whether these extensions are already expressible, and ideally, prove that they are (not).

In the case of DASH, it turns out that many of the constraints are expressible if we allow for *full* equality constraints. This is a fundamental addition to SHACL, as shown in Chapter 4. A natural conclusion is to simply add full equality to the SHACL recommendation. In real SHACL, this means allowing full property paths as a parameter for the equality constraint component. More generally, this idea can be applied to all the property pair constraint components. This then also solves the issue with *lessThan* and *lessThanEq* discussed in Remark 2.13: $\mathit{moreThan}(E_1, E_2)$ is equivalent with $\mathit{lessThan}(E_2, E_1)$, and similarly, $\mathit{moreThanEq}(E_1, E_2)$ is equivalent with $\mathit{lessThanEq}(E_2, E_1)$.

6.4 Provenance

In Chapter 5, we have proposed a provenance semantics for SHACL. In addition to the desirability of supporting provenance from a general database perspective, the utility of a provenance semantics for shapes to support data footprint in Linked Data applications has

[1] Their approach is to postulate two shape names s_1 and s_2 that can be assigned arbitrary sets of nodes, as long as the two sets form a partition of the domain. Then for one node x to satisfy the shape $\mathit{disj}(E, p)$, it is sufficient that $E(x)$ is a subset of s_1 and $p(x)$ of s_2. This condition is not necessary, however, as other nodes may require different partitions.

been pointed out informally by prominent Semantic Web researchers [14,101]. Moreover, the idea of using shapes to *describe* nodes in a graph has been floating around in the community [89]. Also, the SHACL Recommendation itself anticipates applications for shapes beyond conformance checking. Our work serves to put these ideas on a firm formal footing.

Our notions of shape fragment serve to open up SHACL: initially conceived as a constraint or data validation language, it can now also serve as a data retrieval language. If shapes are available, either in a schema coming from the producer of the data, or as an expression of an application's interest in certain types of information, they can now be used to retrieve data. In such settings we avoid the need to switch to a separate retrieval language, typically SPARQL in this context.

Our approach to defining neighborhoods has been to be deterministic, and to satisfy Sufficiency, while also omitting needless triples, i.e., trying to be minimal. However, as discussed by Glavic [47, Section 2.1], minimality is a requirement for provenance semantics that is challenging, and sometimes impossible to achieve together with determinism and Sufficiency. See also our Remark 5.7 in Section 5.1.4.

6.5 Research directions

SHACL should be understood in context of other related technologies. Within the Semantic Web technologies stack, this means understanding its place in relation to SPARQL and OWL. In the case of OWL, Chapter 2 helps towards this end: we give a purely logical formulation of SHACL that resembles the logical foundations of OWL. There are still questions here, for example, how to *combine* SHACL and OWL, i.e., what does it mean to validate an RDF graph in the presence of an ontology? This direction is being explored by Ahmetaj et al. [4]. In the case of SPARQL, Corman et al. [29] showed the connections between (recursive) SHACL and SPARQL: SHACL shapes can be expressed as unary queries using SPARQL. However, there is also another shape language for RDF, namely, ShEx [23]. It similarly has a concept of 'shapes' as unary queries and the main task is also validation. However, the design of ShEx differs fundamentally from SHACL. The main difference is the use of regular expressions in shapes to describe the immediate neighborhood of nodes, given by the outgoing and incoming properties. Similarly to SHACL, there is also an emphasis on the navigational aspect of shapes by using recursion as opposed to regular path expressions in SHACL. There are two interesting research tasks here: finding a common fragment of the two languages, and defining a combined shape language that encapsulates both SHACL and ShEx. Given the different designs of the languages, these are challenging tasks.

Going further, looking at RDF and SHACL from a graph database perspective, it is also of interest to develop a further understanding of the relation between SHACL and PG-Schema. PG-Schema is the proposed schema language for the property graph data model [8]. Here, the challenge is to compare languages that have different logical data models. There are two immediately recognizable approaches to this problem: you can study the languages using an intersection of the two data models, or you can take a unifying graph data model like the Domain Graph model from MillenniumDB [103] and try to define SHACL, ShEx and PG-Schema for that setting. Both approaches should be considered and come with their own challenges.

Another topic of research is the extension of SHACL for the complete RDF data model. SHACL distinguishes the IRIs used as nodes from IRIs used as predicate names, even when they are the exact same IRI. For example, the constraint "all triples that have a predicate of type 'HumanProperty' must have a subject of type 'Human' " is not expressible in SHACL. This opens SHACL up for more general shapes about RDF documents that, for example, contain some information on the properties that might be checked. Finding a suitable underlying formal model and understanding its usefulness for expressing shapes in such a setting are possible next steps here.

Finally, there is one last gap in the SHACL recommendation that has not been discussed so far, which are the *validation reports*. The SHACL specification only requires this report to contain the violating nodes, together with the constraint components that they violate. However, this kind of information may be too little for users to understand what is going wrong. Finding suitable explanations for non-validation is therefore of interest, and this was already explored by Ahmetaj et al. [3] where they consider ‘repairs’ (additions and deletions) for the data graph as a possible explanation for non-validation. This line of work has close ties with data provenance and causality in the database literature [21].

Bibliography

[1] D. Abadi et al. The Seattle report on database research. *SIGMOD Record*, 48(4):44–53, 2019.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] S. Ahmetaj, R. David, M. Ortiz, A. Polleres, B. Shehu, and M. ˇ Simkus. Reasoning about explanations for non-validation in SHACL. In Meghyn Bienvenu, Gerhard Lakemeyer, and Esra Erdem, editors, *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, pages 12–21. IJCAI Organization, 2021.

[4] Shqiponja Ahmetaj, Magdalena Ortiz, Anouk Oudshoorn, and Mantas Simkus. Reconciling SHACL and ontologies: Semantics and validation via rewriting. In Kobi Gal, Ann Nowé, Grzegorz J. Nalepa, Roy Fairstein, and Roxana Radulescu, editors, *ECAI 2023 - 26th European Conference on Artificial Intelligence*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, pages 27–35. IOS Press, 2023. `doi:10.3233/FAIA230250`.

[5] G. Aluç, O. Hartig, T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In P. Mika, T. Tudorache, et al., editors, *Proceedings 13th International Semantic Web Conference*, volume 8796 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2014.

[6] M. Andreşel, J. Corman, M. Ortiz, J.L. Reutter, O. Savkovic, and M. ˇ Simkus. Stable model semantics for recursive SHACL. In Y. Huang, I. King, T.-Y. Liu, and M. van Steen, editors, *Proceedings WWW'20*, pages 1570–1580. ACM, 2020.

[7] R. Angles and C. Gutierrez. The expressive power of SPARQL. In A. Sheth, S. Staab, et al., editors, *Proceedings 7th International Semantic Web Conference*, volume 5318 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2008.

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

[9] M. Arenas and J. Pérez. Querying semantic web data with SPARQL. In *Proceedings 30st ACM Symposium on Principles of Databases*, pages 305–316. ACM, 2011.

[10] M. Arenas, J. Pérez, and C. Gutierrez. On the semantics of SPARQL. In R. De Virgilio, F. Giunchiglia, and L. Tanca, editors, *Semantic Web Information Management—A Model-Based Perspective*, pages 281–307. Springer, 2009.

[11] D. Arroyuelo, A. Hogan, G. Navarro, and J. Rojas-Ledesma. Time- and space-efficient regular path queries. In *Proceedings 38th International Conference on Data Engineering*, pages 3091–3105. IEEE, 2022.

[12] F. Baader, I. Horrocks, C. Lutz, and U. Sattler. *An Introduction to Description Logic*. Cambridge University Press, 2017.

[13] P. Barceló, C.A. Hurtado, L. Libkin, and P.T. Wood. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems*, 37(4):31:1–31:46, 2012.

[14] T. Berners-Lee. Linked data shapes, forms and footprints. `https://www.w3.org/DesignIssues/Footprints.html`, 2019.

[15] C. Bizer and A. Schultz. The Berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems*, 5(2):1–24, 2009.

[16] Bart Bogaerts and Luís Cruz-Filipe. Fixpoint semantics for active integrity constraints. *AIJ*, 255:43–70, 2018. `doi:10.1016/j.artint.2017.11.003`.

[17] Bart Bogaerts and Luís Cruz-Filipe. Stratification in approximation fixpoint theory and its application to active integrity constraints. *ACM Transactions on Computational Logic*, 22(1):6:1–6:19, 2021. `doi:10.1145/3430750`.

[18] Bart Bogaerts and Maxime Jakubowski. Fixpoint semantics for recursive SHACL. In A. Formisano, Y.A. Liu, et al., editors, *Proceedings 37th International Conference on Logic Programming (Technical Communications)*, volume 345 of *Electronic Proceedings in Theoretical Computer Science*, pages 41–47, 2021.

[19] Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. Expressiveness of SHACL features and extensions for full equality and disjointness tests. *Log. Methods Comput. Sci.*, 20(1), 2024. `doi:10.46298/LMCS-20(1:16)2024`.

[20] Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. SHACL: A description logic in disguise. In G. Gottlob, D. Inclezan, and M. Maratea, editors, *Logic Programming and Nonmonotonic Reasoning*, pages 75–88. Springer International Publishing, 2022.

[21] Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. Postulates for provenance: Instance-based provenance for first-order logic, 2024. ACM Symposium on Principles of Database Systems, June 2024, to appear.

[22] Bart Bogaerts, Joost Vennekens, and Marc Denecker. Grounded fixpoints and their applications in knowledge representation. *Artificial Intelligence Journal*, 224:51–71, 2015. `doi:10.1016/j.artint.2015.03.006`.

[23] I. Boneva, J.E.L. Gayo, and E.G. Prud'hommeaux. Semantics and validation of shape schemas for RDF. In C. d'Amato, M. Fernandez, V. Tamma, et al., editors, *Proceedings 16th International Semantic Web Conference*, volume 10587 of *Lecture Notes in Computer Science*, pages 104–120. Springer, 2017.

[24] P. Buneman, S. Khanna, and W.C. Tan. Why and where: A characterization of data provenance. In J. Van den Bussche and V. Vianu, editors, *Database Theory—ICDT 2001*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2001.

[25] D. Calvanese, G. De Giacomo, D. Nardi, and M. Lenzerini. Reasoning in expressive description logics. In F. Baader, D. Calvanese, D. McGuiness, D. Nardi, and P. Patel-Schneider, editors, *The Description Logic Handbook*, chapter 23. Cambridge University Press, 2003.

[26] Angelos Charalambidis, Panos Rondogiannis, and Ioanna Symeonidou. Approximation fixpoint theory and the well-founded semantics of higher-order logic programs. *Theory and Practice of Logic Programming*, 18(3-4):421–437, 2018. `doi:10.1017/S1471068418000108`.

[27] J. Cheney, L. Chiticarius, and W.-C. Tan. Provenance in databases: why, how and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[28] Adrian Chmuroviĉ and Mantas ˇ Simkus. Well-founded semantics for recursive SHACL. In *Datalog-2.0 2022, 4th International Workshop on the Resurgence of Datalog in Academia and Industry*, CEUR Workshop proceedings, pages 2–13, 2022.

[29] J. Corman, F. Florenzano, J.L. Reutter, and O. Savkovic. Validating SHACL constraints over a SPARQL endpoint. In C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, et al., editors, *Proceedings 18th International Semantic Web Conference*, volume 11778 of *Lecture Notes in Computer Science*, pages 145–163. Springer, 2019.

[30] J. Corman, J.L. Reutter, and O. Savkovic. Semantics and validation of recursive SHACL. In D. Vrandecic et al., editors, *Proceedings 17th International Semantic Web Conference*, volume 11136 of *Lecture Notes in Computer Science*, pages 318–336. Springer, 2018. Extended version, technical report KRDB18-01, `https://www.inf.unibz.it/krdb/tech-reports/`.

[31] Y. Cui, J. Widom, and J.L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2):179–227, 2000.

[32] DBLP data in RDF. `http://dblp.org/rdf/`.

[33] Thomas Delva, Anastasia Dimou, Maxime Jakubowski, and Jan Van den Bussche. Data provenance for SHACL. In J. Stoyanovich, J. Teubner, et al., editors, *Proceedings 26th International Conference on Extending Database Technology*, pages 285–297. openproceedings.org, 2023.

[34] Marc Denecker, Victor Marek, and Mirosław Truszczyński. Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. *Logic-Based Artificial Intelligence*, pages 127–144, 2000. `doi:10.1007/978-1-4615-1567-8_6`.

[35] Marc Denecker, Victor Marek, and Mirosław Truszczyński. Uniform semantic treatment of default and autoepistemic logics. *Artificial Intelligence Journal*, 143(1):79–122, 2003. `doi:10.1016/S0004-3702(02)00293-X`.

[36] Marc Denecker, Victor Marek, and Mirosław Truszczyński. Ultimate approximation and its application in nonmonotonic knowledge representation systems. *Information and Computation*, 192(1):84–121, 2004. `doi:10.1016/j.ic.2004.02.004`.

[37] Marc Denecker and Joost Vennekens. The well-founded semantics is the principle of inductive definition, revisited. In Chitta Baral, Giuseppe De Giacomo, and Thomas Eiter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the 14th International Conference*, pages 1–10. AAAI Press, 2014.

[38] M. Eich, P. Fender, and G. Moerkotte. Efficient generation of query plans containing group-by, join, and groupjoin. *The VLDB Journal*, 27(5):617–641, 2018.

[39] Herbert Enderton. *A mathematical introduction to logic.* Academic Press, 2nd edition, 2001.

[40] Nicolas Ferranti, Jairo Francisco de Souza, Shqiponja Ahmetaj, and Axel Polleres. Formalizing and validating wikidata's property constraints using shacl and sparql. *Semantic Web Journal*, 2024. under review.

[41] M. Figuera, Ph.D. Rohde, and M.-E. Vidal. Trav-SHACL: Efficiently validating networks of SHACL constraints. In J. Leskovec et al., editors, *Proceedings WWW'21*, pages 3337–3348. ACM, 2021.

[42] Melvin Fitting. Fixpoint semantics for logic programming — A survey. *Theoretical Computer Science*, 278(1-2):25–51, 2002. `doi:10.1016/S0304-3975(00)00330-3`.

[43] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In SIGMOD [88], pages 1433–1445.

[44] J.E.L. Gayo, H. Knublauch, and D. Kontokostas. SHACL test suite and implementation report. `https://w3c.github.io/data-shapes/data-shapes-test-suite/`, January 2021.

[45] J.E.L. Gayo, E. Prud'hommeaux, I. Boneva, and D. Kontokostas. Validating RDF data. *Synthesis Lectures on the Semantic Web: Theory and Technology*, 16, 2018.

[46] Jose Emilio Labra Gayo, Eric Prud'hommeaux, Iovka Boneva, and Dimitris Kontokostas. *Validating RDF Data.* Springer International Publishing, 2018. `doi:10.1007/978-3-031-79478-0`.

[47] Boris Glavic. Data provenance: Origins, applications, algorithms, and models. *Foundations and Trends in Databases*, 9(3–4):209–441, 2021. `doi:10.1561/1900000068`.

[48] E. Grädel and V. Tannen. Semiring provenance for first-order model checking. arXiv:1712.01980, 2017.

[49] T.J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings 26th ACM Symposium on Principles of Database Systems*, pages 31–40, 2007.

[50] S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C Recommendation, March 2013.

[51] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proceedings 23rd International Conference on Very Large Data Bases*, pages 386–395. Morgan Kaufmann, 1997.

[52] I. Horrocks, O. Kutz, and U. Sattler. The even more irresistible SROIQ. In Chitta Baral, James P. Delgrande, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the 15th International Conference*, pages 57–67. AAAI Press, 2016.

[53] Maxime Jakubowski. SHACL Logical Syntax Parser, January 2024. URL: `https://github.com/MaximeJakubowski/sls_project`, `doi:10.5281/zenodo.10462613`.

[54] Maxime Jakubowski. SPARQL RML Rewriter, January 2024. URL: `https://github.com/MaximeJakubowski/SRR`, `doi:10.5281/zenodo.10462628`.

[55] Holger Knublauch. SHACL and OWL compared. `https://spinrdf.org/shacl-and-owl.html`. Accessed: 2021-06-16.

[56] Holger Knublauch. Dash constraint components. `https://datashapes.org/constraints.html`, 2021.

[57] S. Köhler, B. Ludäscher, and D. Zinn. First-order provenance games. In V. Tannen, L. Wong, et al., editors, *In Search of Elegance in the Theory and Practice of Computation*, volume 8000 of *Lecture Notes in Computer Science*, pages 382–399. Springer, 2013.

[58] Ph. G. Kolaitis. On the expressive power of logics on finite models. In *Finite Model Theory and Its Applications*, chapter 2. Springer, 2007.

[59] J.E. Labra Gayo. Creating knowledge graph subsets using shape expressions. arXiv:2110.11709, October 2021.

[60] LDBC Graph Query Language Task Force. G-CORE: A core for future graph query languages. In SIGMOD [88], pages 1421–1432.

[61] Martin Leinberger, Philipp Seifer, Tjitze Rienstra, Ralf Lämmel, and Steffen Staab. Deciding SHACL shape containment through description logics reasoning. In Pan et al. [68], pages 366–383.

[62] Fangfang Liu, Yi Bi, Md. Solimul Chowdhury, Jia-Huai You, and Zhiyong Feng. Flexible approximators for approximating fixpoint theory. In *Proceedings of Canadian AI*, pages 224–236, 2016. `doi:10.1007/978-3-319-34111-8_28`.

[63] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 157–168, 2003.

[64] G. Moerkotte and Th. Neumann. Accelerating queries with group-by and join by groupjoin. *Proceedings of the VLDB Endowment*, 4:843–851, 2011.

[65] Magdalena Ortiz. A short introduction to shacl for logicians. In Helle Hvid Hansen, Andre Scedrov, and Ruy J.G.B. de Queiroz, editors, *Logic, Language, Information, and Computation*, pages 19–32. Springer Nature Switzerland, 2023.

[66] M. Otto. *Bounded Variable Logics and Counting: A Study in Finite Models*, volume 9 of *Lecture Notes in Logic*. Springer, 1997.

[67] OWL 2 Web ontology language: Structural specification and functional-style syntax. W3C Recommendation, December 2012.

[68] J.Z. Pan et al., editors. *Proceedings 19th International Semantic Web Conference*, volume 12506 of *Lecture Notes in Computer Science*. Springer, 2020.

[69] P. Pareti, G. Konstantinidis, and F. Mogavero. Satisfiability and containment of recursive shacl. *Journal of Web Semantics*, 74:100721, 2022. `doi:10.1016/j.websem.2022.100721`.

[70] P. Pareti, G. Konstantinidis, F. Mogavero, and T. J. Norman. SHACL satisfiability and containment. In Pan et al. [68], pages 474–493.

[71] L.D. Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, 2007.

[72] Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*, 7(3):301–353, 2007. `doi:10.1017/S1471068406002973`.

[73] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):article 16, 2009.

[74] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *Journal of Web Semantics*, 8(4):255–270, 2010.

[75] Irena Polikoff. Why I don't use OWL anymore – Top Quadrant blog. `https://www.topquadrant.com/owl-blog/`. Accessed: 2021-06-04.

[76] RDFLib/pySHACL: A Python validator for SHACL. `https://github.com/RDFLib/pySHACL`, May 2021.

[77] pySHACL-fragments software. `https://github.com/shape-fragments/pySHACL-fragments`.

[78] RDF 1.1 concepts and abstract syntax. W3C Recommendation, February 2014.

[79] RDF 1.1 primer. W3C Working Group Note, June 2014.

[80] RDF 1.1 semantics. W3C Recommendation, February 2014.

[81] RDF 1.1 turtle. W3C Recommendation, February 2014.

[82] J. Reutter. Personal communication, 15 January 2021.

[83] Robert Schaffenrath, Daniel Proksch, Markus Kopp, Iacopo Albasini, Oleksandra Panasiuk, and Anna Fensel. Benchmark for performance evaluation of shacl implementations in graph databases. In *International Joint Conference on Rules and Reasoning*, pages 82–96. Springer, 2020.

[84] M. Schmidt-Schauß. Subsumption in KL-ONE is undecidable. In R. J. Brachman, editor, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, volume 1 of *Representation and Reasoning*, pages 421–431, 1989.

[85] Philipp Seifer, Daniel Hernández, Ralf Lämmel, and Steffen Staab. From shapes to shapes: Inferring SHACL shapes for results of SPARQL CONSTRUCT queries (extended version). *CoRR*, abs/2402.08509, 2024. `arXiv:2402.08509`, `doi:10.48550/ARXIV.2402.08509`.

[86] Shapes constraint language (SHACL). W3C Recommendation, July 2017.

[87] `https://github.com/shexjs/shex.js`.

[88] *Proceedings 2018 International Conference on Management of Data*. ACM, 2018.

[89] SPARQL 1.2 community group. DESCRIBE using shapes. `https://github.com/w3c/sparql-12/issues/39`.

[90] Hannes Strass. Approximating operators and semantics for abstract dialectical frameworks. *Artificial Intelligence Journal*, 205:39–70, 2013. doi:10.1016/j.artint.2013.09.004.

[91] V. Tannen. Provenance analysis for FOL model checking. *ACM SIGLOG News*, 4(1):24–36, 2017.

[92] TopQuadrant. An overview of SHACL: A new W3C standard for data validation and modeling. https://www.topquadrant.com/an-overview-of-shacl/, 2017. Webinar slides.

[93] M. Truszczynski. An introduction to the stable and well-founded semantics of logic programs. In M. Kifer and Y.A. Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, pages 121–177. ACM and Morgan & Claypool, 2018.

[94] Mirosław Truszczyński. Strong and uniform equivalence of nonmonotonic theories - an algebraic approach. *Annals Mathematics and Artificial Intelligence*, 48(3-4):245–265, 2006. doi:10.1007/s10472-007-9049-2.

[95] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, 1988.

[96] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976. doi:10.1145/321978.321991.

[97] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991. doi:10.1145/116825.116838.

[98] Maarten Vandenbrande, Maxime Jakubowski, Pieter Bonte, Bart Buelens, Femke Ongenae, and Jan Van den Bussche. POD-QUERY: schema mapping and query rewriting for solid pods. In Irini Fundulaki, Kouji Kozaki, Daniel Garijo, and José Manuél Gómez-Pérez, editors, *Proceedings of the ISWC 2023 Posters, Demos and Industry Tracks*, volume 3632 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2023.

[99] Joost Vennekens, David Gilis, and Marc Denecker. Splitting an operator: Algebraic modularity results for logics with fixpoint semantics. *ACM Transactions on Computational Logic*, 7(4):765–797, 2006. doi:10.1145/1182613.1189735.

[100] Joost Vennekens, Maarten Mariën, Johan Wittocx, and Marc Denecker. Predicate introduction for logics with a fixpoint semantics. Parts I and II. *Fundamenta Informaticae*, 79(1-2):187–227, 2007. doi:10.5555/2366527.2366536.

[101] R. Verborgh. Shaping linked data apps. https://ruben.verborgh.org/blog/2019/06/17/shaping-linked-data-apps/, 2019.

[102] R. Verborgh, M. Vander Sande, O. Hartig, et al. Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. *Journal of Web Semantics*, 37–38:184–206, 2016.

[103] Domagoj Vrgoc, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. Millenniumdb: A persistent, open-source, graph database. *CoRR*, abs/2111.01540, 2021. URL: https://arxiv.org/abs/2111.01540, arXiv:2111.01540.