A Review of SHACL: From Data Validation to Schema Reasoning for RDF Graphs

Paolo Pareti[1] and George Konstantinidis[2]

[1] University of Winchester, Winchester, United Kingdom
paolo.pareti@winchester.ac.uk
[2] University of Southampton, Southampton, United Kingdom
g.konstantinidis@soton.ac.uk

Abstract. We present an introduction and a review of the *Shapes Constraint Language* (SHACL), the W3C recommendation language for validating RDF data. A SHACL document describes a set of constraints on RDF nodes, and a graph is valid with respect to the document if its nodes satisfy these constraints. We revisit the basic concepts of the language, its constructs and components and their interaction. We review the different formal frameworks used to study this language and the different semantics proposed. We examine a number of related problems, from containment and satisfiability to the interaction of SHACL with inference rules, and exhibit how different modellings of the language are useful for different problems. We also cover practical aspects of SHACL, discussing its implementations and state of adoption, to present a holistic review useful to practitioners and theoreticians alike.

1 Introduction

The Shapes Constraint Language (SHACL) [23] is a W3C recommendation language for the validation of RDF graphs. In SHACL, validation is based on *shapes*, which define particular constraints and specify which nodes in a graph should be validated against these constraints. The ability to validate data with respect to a set of constraints is of particular importance for RDF graphs, as they are schemaless by design. Validation can be used to detect problems in a dataset and it can provide data quality guarantees for the purpose of data exchange and interoperability. A set of constraints can also be interpreted as a "schema", functioning as one of the primary descriptors of a graph dataset, thus enhancing its understandability and usability. A set of SHACL shapes is called a *shapes graph*, but we refer to it as a SHACL *document* in order not to confuse it with the graphs that it is used to validate.

In this article we present a review of SHACL, which is composed of three main parts. In the first part, in Sections 2 and 3, we review the SHACL specification. This part focuses on how shapes are defined, and how they are used for the purpose of validation. We highlight the main peculiarities of this language, and discuss how SHACL validation can be expressed either in terms of SPARQL queries, to facilitate its implementation, or in terms of *assignments* [13], to make it

amenable to theoretical study. The syntax of SHACL is outside the scope of this review, and for the precise details on how to encode particular constraints we refer the reader to the SHACL specification [23]. We also do not discuss the process that lead to the development of SHACL, but it should be noted that this specification was built on top of a number of previous constraint languages, the most influential of which is Shape Expressions (ShEx) [48].

In the second part of our review, in Sections 5 to 8, we present the formal properties of this language. This mainly revolves around a discussion of *recursion*. The semantics of recursion is not defined in the SHACL specification, and thus has been the subject of significant subsequent research [3]. The formal semantics of SHACL is given as a translation into `SCL` [37], a first order logic language that captures the entirety of the SHACL specification. Apart from validation, several standard decision problems are discussed, such as *satisfiability* and *containment*, along with an existing study on the interaction of SHACL with inference rules. We try to keep a consistent notation throughout this article and at times this notation might be different from the one in the original articles.

In Section 9, we review existing implementations of SHACL validators and their integration with mainstream graph databases. We also review prominent additional tools to manage SHACL documents, such as tools designed to automate or semi-automate the process of creating SHACL documents by exploiting graph data, ontologies, or other constraint languages. These approaches provide solutions to the cold start problem, and alleviate reliance on expert knowledge, which are typical problems of new technologies. We complement a discussion of these approaches with a review of prominent applications of SHACL in several domains; in summary, the abundance of SHACL related tools and applications highlights the remarkable level of maturity and adoption reached by this relatively new language.

2 Preliminaries

Before discussing SHACL, we briefly introduce our notation for RDF graphs [14]. With the term RDF *graph* (or just *graph*) we refer to a set of RDF *triples* (or just *triples*), where each triple $<s, p, o>$ identifies an edge with label p, called *predicate*, from a node s, called *subject*, to a node o, called *object*. Subjects, predicates and objects of RDF triples are collectively called RDF *terms*. The RDF terms that appear as subject and objects in the triples of a graph are called the *nodes* of the graph. Graphs in this article are represented in Turtle syntax using common XML namespaces, such as `sh`, `rdf` and `rdfs` to refer to, respectively, the SHACL, RDF, and RDFS [7] vocabularies. Queries over RDF graph will be expressed as SPARQL [42] queries.

In the RDF data model, subjects, predicates and objects are defined over different but overlapping domains. For example, while RDF terms of the IRI type can occupy any position in a triple, RDF terms of the *literal* type (representing datatype values) can only appear in the object position. These differences are not central to the topics discussed in this review, and thus, for the sake of simplifying

```
:EmployeeShape a sh:PropertyShape ;
    sh:targetClass :Employee ;
    sh:path :hasOfficeNumber ;
    sh:minCount 1 .
```

```
:Anne a :Employee .
:Bob a :Employee ;
    :hasOfficeNumber "18" ;
    :hasOfficeNumber "3" .
:Carl a :Employee ;
    :hasOfficeNumber "171" .
:David a :Customer .
```

Fig. 1. (Left) A sample SHACL document (shape graph) stating the constraint that every employee must have at least one office number. (Right) A sample RDF graph (data graph).

notation, we will assume that all elements of a triple are drawn from a single and infinite domain of constants. This corresponds to the notion of *generalized* RDF [14].

3 Overview of SHACL

The main application of SHACL is data validation. Data validation in SHACL requires two inputs: (1) an RDF graph G to be validated and (2) a SHACL document M that defines the conditions against which G must be evaluated. The SHACL specification defines the output of the data validation process as a *validation report*, detailing all the violations that were found in G of the conditions set by M. If the violation report contains no violations, a graph G is *valid* w.r.t. SHACL document M. The SHACL validation process can be abstracted into the following decision problem. Given a graph G and a SHACL document M, we denote with $\textsc{Validate}(\mathsf{G}, \mathsf{M})$ the decision problem of deciding whether G is *valid* w.r.t. SHACL document M, that we call *validating* G against M.

For example, the graph on the left of Figure 1 represents a SHACL document M_1, that defines the condition that every employee must have an office number. Therefore, the validation report for a graph and M_1 would list all of the instances of `:Employee` in the graph that do not have an office number. The validation report for M_1 and the data graph G_1 on the right of Figure 1 contains a violation on node `rdf:Anne`, since she does not have an office number. Therefore G_1 is not valid w.r.t. M_1.

Formally, a SHACL *document* is a set of *shapes*. Validating a graph against a SHACL document involves validating it against each shape. Shapes restrict the structure of a valid graph by focusing on certain nodes and examining whether they satisfy their constraints. The main components of a shape are a *constraint* d and a *target definition* t. Constraints can be *evaluated* on any RDF node to determine whether that node *satisfies* or not the given constraints. A node that satisfies the constraint of a shape it is said to *conform* to that shape, or *not-conform* otherwise. If a shape has an empty constraint, all nodes trivially conform to the shape. Not all nodes of a graph must conform to all the shapes in the

SHACL document. The constraint definition of each shape defines which RDF nodes, called *target nodes*, must conform to that shape in order for the graph to be valid. A shape with an empty constraint definition does not have any target nodes. Through inter-shape referencing, as we will see below, additional nodes might be required to conform to certain shapes (or not, if negation is used) for the validation to succeed. Further irrelevant nodes within the graph do not play a role in validation of the shape, whether they conform to it or not. The SHACL document M_1 of our previous example, contains shape :`EmployeeShape`, whose constraint captures the property of "having an office number", and whose target definition targets only the RDF nodes of type Employee. Nodes of the Client type do not generate violations by not having an office number.

Formally, a shape is a tuple $\langle \mathtt{s}, t, d \rangle$ defined by three components: (1) the shape name `s`, which uniquely identifies the shape; (2) the *target definition* t, and (3) the set of *constraints* which are used in conjunction, and hence hereafter referred to as the single constraint d. As demonstrated in Figure 1, a SHACL document is itself an RDF graph. The graph representing a SHACL document is called a *shapes graph*, while the graph being validated is called a *data graph*. This approach to serialisation is similar to how OWL ontologies are serialised, and it serves a similar purpose. Thanks to this approach, a SHACL document does not require any dedicated infrastructure to be stored and shared. In fact, a SHACL document can be embedded directly into the very graph it validates, thus combining the shape graph and data graph into a single graph. Interestingly, with this serialisation, a shapes graph, being an RDF graph, can be itself subject to validation. The SHACL specification, in fact, defines a shapes graph that can be used to validate shapes graphs.

We will now look in more details at the two major components of shapes, namely target definitions and constraints.

3.1 SHACL Target Definitions

A SHACL target definition, within a constraint, is a set of *target declarations*. There are four types of target declarations defined in SHACL, each one taking an RDF term `c` as a parameter.

Node Targets A node target declaration on `c` targets that specific node.

Class-based Targets If a shape has a class-based target on `c`, then all the nodes in the graph that are of type (`rdf:type`) `c` are target nodes for that shape.

Subjects-of Targets If a shape has a subject-of target on `c`, then the target nodes for that shape are all the nodes in the graph that appear as subjects in triples with `c` as the predicate.

Objects-of Targets If a shape has an object-of target on `c`, then the target nodes for that shape are all the nodes in the graph that appear as objects in triples with `c` as the predicate.

The shape defined in Figure 1 demonstrates an example of a class-based target targeting class :`Employee`. Similarly, to target a subject of a property, e.g., :`worksAt`, the second line of the shape definition would be substituted with:

```
sh:targetSubjectOf :worksAt.
```

Typically, a target declaration is used to select, among all the nodes in a graph, the ones to target for constraint validation. The *node target* declaration, however, behaves differently, as it targets a particular node regardless of whether this node occurs in the graph or not. An important implication of this is that empty graphs are not trivially valid, since node targets can detect violations on nodes external to the graph. If a target definition of a shape is empty, then that shape will have no target nodes. However, this does not mean that the constraint of that shape will not be evaluated on any nodes since, as mentioned, other shapes can refer to it and "pass it" a node to check for conformance.

3.2 Focus nodes and property paths

When a target or another node is considered against a shape for conformity, we call it a *focus* node. Initially a shape focuses on its target nodes (these are the initial set of focus nodes). Additional focus nodes are obtained by following SHACL *property paths*, which we also refer to as just *paths*. SHACL property paths are a subset of SPARQL *property paths* and, as the name suggests, define paths in the RDF graph. The simplest type of path, called predicate path, corresponds to a single property IRI `c`. This path identifies all the nodes that are reachable in the RDF graph from the current focus node by following a single edge `c`. In other words, this path identifies all the RDF nodes in the object position of triples that have `c` as the predicate and the current focus node as the subject. More complex paths can be constructed by inverting the direction of a path, by concatenating two different paths one after the other, or by allowing the repetition of a path for a minimum, maximum or arbitrary number of times.

Based on the use of property paths, SHACL specification distinguishes shapes into two types: *node shapes* and *property shapes*. Intuitively, the constraint of a node shape is evaluated directly on the focus nodes of the shape. Instead, when using a property path, shapes must be declared as a property shapes. These are characterised by a path, and their constraints are evaluated over all of the nodes that can be reached from the focus nodes following such path. For example, the constraint that every employee's password must be at least 8 characters long can be represented by a property shape that targets employee nodes, and that has a relation such as `:hasPassword` as its path. In this way, the actual nodes that must satisfy the "at least 8 characters long" constraint are not the target nodes, but instead those that appear as objects in triples with an employee node as a subject, and `:hasPassword` as the predicate.

3.3 SHACL Constraints

The majority of the SHACL recommendation is dedicated to defining the different types of constraint components that can be used in SHACL constraints. The main type of constraint components are called *core constraint components*. These are the components that SHACL compliant systems typically support, and where

most of the existing literature focuses on. The other main type of components are the SPARQL-*based constraint components*, that are used to embed SPARQL queries into SHACL constraints. This significantly increases the expressive power of such constraints. However, the inclusion of arbitrarily complex SPARQL queries can lead to performance issues, and can make such constraints harder to understand and use. It is also worth noting that, outside of the SHACL recommendation, a number of additional SHACL features[3] are currently being designed, and some of them might be included in further versions of SHACL. In the rest of this paper we will focus on core constraint components.

In order to better understand SHACL core constraint components, we propose a broad categorisation of these components into three main categories, depending on how they are evaluated on the focus nodes. Notice that most constraint components can be used in both node shapes and property shapes.

Graph Structure Components. These components define constraints that are evaluated at the level of triples of the graph, and focus on restrictions such as the minimum and maximum cardinality that the focus node must have for certain paths, or the RDF class that the focus node should be a type of. The shape defined in Figure 1 demonstrates an example of a minimum cardinality constraint for predicate path :`hasOfficeNumber`. Two other salient constraints in this category are the property pair equality and disjointedness, that specify whether the two sets of nodes reachable from two different paths must be equal or disjoint, respectively.

Filter Components. These components define constraints that are evaluated at the level of nodes, and their evaluation is usually independent from the triples present in the graph. Filter constraints restrict the focus node (1) to be a particular RDF term, (2) to be of a particular type, such as IRI, blank node or literal, or (3) to be a literal that satisfies certain properties, such as being of the integer datatype, or a string produced by a certain regular expression.

Logical Components. Logical components define the standard logical operators of conjunction, disjunction and negation over other constraints.

While most core constraint components fall into one of these categories, the pair of constraints `sh:lessThan` and `sh:lessThanOrEquals` is a notable exception, as it is combines the properties of graph structure and filter components. These two constraints require all the nodes reachable by one path to be literals that are less than (resp. less than or equals) to the nodes reachable by a second path.

It is worth noting that all constraints but one, namely `sh:closed`, are not affected by triples with unknown predicates (i.e. predicates not occurring in the SHACL document). This means that if a graph is valid with respect to a set of those constraints, it would still remain valid if new triples with unknown predicates are added to the graph. Thus, given a non-empty graph G, valid w.r.t. a SHACL document M, graph $G \cup <s, p, o>$ is also valid w.r.t. M if (1) p does not occur in M and (2) M does not contain the `sh:closed` constraint component.

[3] `https://w3c.github.io/shacl/shacl-af/` accessed on 18/6/21

```
:EmployeeShapeB a sh:PropertyShape ;
   sh:targetClass :Employee ;
   sh:path :hasOfficeNumber ;
   sh:qualifiedMinCount 1 ;
   sh:qualifiedValueShape :OfficeNumberShape .

:OfficeNumberShape a sh:NodeShape ;
   sh:minLength 3 .
```

Fig. 2. A sample SHACL document stating the constraint that every employee must have at least one 3-characters or longer office number.

Intuitively, this means that those constraints restrict the usage of terms from a particular vocabulary, but they do not restrict in any way the graph from containing triples described using other vocabularies. The `sh:closed` component, on the other hand, restricts the predicates of the triples that have the focus node as a subject to belong to a predetermined finite set. Effectively, the `sh:closed` component can prohibit the use of unknown predicate relations for certain nodes in the graph, and thus prevent the inclusion of terms from other vocabularies. Interestingly, component `sh:closed` introduces an asymmety in SHACL, since it only affects triples where the focus node is the subject, and it is not possible to define a similar constraint for nodes in the object position.

A major feature of SHACL is that constraints can use the name of a shape to require a particular set of nodes to conform to that shape. This is called a *shape reference*. An example of a shape reference is demonstrated by the SHACL document in Figure 2. This document contains shape `:EmployeeShapeB` which references shape `:OfficeNumberShape`. The former shape restricts all of its target nodes to having an edge `:hasOfficeNumber` to a node that conforms to the latter shape, having a string length of at least three characters. Validating the data graph in Figure 1 with the SHACL document in Figure 2 results in two violating nodes for shape `:EmployeeShapeB`. The first one is `:Anne`, who does not have an office number, and the second one is `:Bob`, whose office numbers all contain fewer than three digits.

Shape references can be recursive, that is, the constraint of a shape can reference the constraints of a second shape which, in turn, can reference the constraints of a third shape, and so on, creating a loop. Let S_0^d be the set of all the shape names occurring in a constraint d of a shape $\langle \mathtt{s}, t, d \rangle$; these are the *directly* referenced shapes of $\mathtt{s}$. Let S_{i+1}^d be the set of shapes in S_i^d union the directly referenced shapes of the constraints of the shapes in S_i^d .

Definition 1. *A shape* $\langle \boldsymbol{s}, t, d \rangle$ *is* recursive *if* $\boldsymbol{s} \in S_\infty^d$*; else it is* non-recursive*.*

Definition 2. *A* SHACL *document* M *is* recursive *if it contains a recursive shape, and* non-recursive *otherwise.*

The semantics of recursive SHACL documents are not defined in the SHACL specification. In Section 4 we review the official semantics of non-recursive SHACL

documents, while in Section 5 we review the extended semantics for recursive SHACL document that have been proposed in the literature.

4 SHACL Validation

In this section we present the semantics of SHACL data validation, that is, the VALIDATE(G, M) decision problem, for any given graph G and SHACL document M. In Section 4.1 we review how validation is defined in the SHACL specification, with the help of SPARQL queries. While this query-based description of SHACL semantics can be easily translated into a concrete implementation, it does not lend itself well to theoretical investigation. In Section 4.2 we will discuss an alternative approach to defining SHACL semantics that is instead amenable to a formal study.

4.1 SHACL Validation by SPARQL queries

The validation of an RDF graph G against a SHACL document M can be performed on a shape-by-shape basis. For each shape $\langle \mathtt{s}, t, d \rangle$, this process involves verifying the fact that every node n, targeted by target definition t, satisfies constraint d. Intuitively, graph G is valid w.r.t. M if and only if this fact is true for every shape in M.

Given a graph G and a target definition t, the set of target nodes for t can be computed by evaluating a SPARQL query on G for each target declaration in t, and taking the union of the values returned by these queries. Table 1 details the corresponding SPARQL query for each of the four types of target declarations defined in SHACL. It should be noted that, by default, SHACL does not enforce any particular entailment regime. If an entailment regime is being adopted, then this should be taken into account when developing a SHACL validator. For example, if the RDFS entailment regime [7] is being considered, subclass inference should be accounted for when computing the set of entities of a given class. To accommodate for this entailment regime, the query for the node target in Table 1 could be updated to the following one.

```
SELECT ?x WHERE {
  ?x rdf:type/rdfs:subClassOf* c
}
```

Once an RDF term has been identified as being in the target of a shape, evaluating whether it conforms to the shape can be done using SPARQL queries. In the SHACL specification, in fact, several core constraint components are defined with respect to SPARQL queries. Most notably, the semantics of SHACL filter components is in direct dependence to the semantics of SPARQL filter functions. For example, the `sh:minLength` constraint component restricts a focus node to having a string length equal or larger than a given number. Formally, a focus node n has a `sh:minLength` of j if and only if the following SPARQL query evaluates to true.

Table 1. Target declarations and their corresponding SPARQL queries to compute the set of target nodes on a given graph

Target declaration	SPARQL query
Node target (node `c`)	SELECT ?x WHERE { VALUES ?x { `c` } }
Class target (class `c`)	SELECT ?x WHERE { ?x `rdf:type c` . }
Subjects-of target (predicate `c`)	SELECT ?x WHERE { ?x `c` ?y . }
Objects-of target (relation `c`)	SELECT ?x WHERE { ?y `c` ?x . }

```
ASK {
  FILTER (STRLEN(str(n)) >= j) .
}
```

Not all SHACL constraints, however, can be easily verified by a single SPARQL query. Evaluating whether a constraint that contains shape references is satisfied by a focus node, in fact, might involve evaluating whether other constraints are satisfied by other nodes which, in turn, might require even further constraint evaluations. For example, in order to evaluate whether node `rdf:Carl` from the data graph in Figure 1 conforms to shape `rdf:EmployeeShapeB` from Figure 2, we would need to evaluate whether his office number, namely RDF term "171", conforms to shape `rdf:OfficeNumberShape`. This is especially problematic in case of recursion, as it could generate an infinite series of constraint evaluations. For non-recursive SHACL documents, Corman et al. [12] showed that it is always possible to check the validity of a graph using a single SPARQL query. For example, a graph can be checked against the SHACL document of Figure 2 by evaluating the following SPARQL query.

```
SELECT ?x WHERE {
  ?x a :Employee .
  FILTER NOT EXISTS {
    ?x :hasOfficeNumber ?y .
    FILTER (STRLEN(str(?y)) >= 3) .
  }
}
```

This query selects all RDF nodes of type Employee that do not have an office number with at least three characters. Thus, any RDF term returned by this query is a node violating a shape of the SHACL document. If this query evaluates to an empty set, then the graph that it is evaluated on is valid with respect to the SHACL document.

4.2 Shape Assignments: A Tool for Defining SHACL Validation

The SPARQL-based approach to SHACL validation does not provide a concise and formal description of SHACL semantics. Moreover, it does not provide us with a terminating procedure to check graphs in the face of SHACL recursion. In this

section we review the concept of *shape assignments* (or just *assignments*) [13], which can be used to address the above mentioned problems.

As defined in Table 1, a target declaration t is a unary query over a graph G. We denote with $G \models t(n)$ that a node n is *in the target* of t with respect to a graph G. If t is empty, no node in any graph is in the target of t. The definition of whether a node conforms to a shape, as we previously discussed, does not only depend on the graph G, but it might also depend, due to shape references, on whether other nodes conform to other shapes. Intuitively, the concept of *assignments* [13] is used to keep track, for every RDF node, of all the shapes that it conforms to, and all of those that it does not. Given a document M and a graph G, we denote $\mathsf{nodes(G, M)}$ the set of nodes in G together with any extra ones referenced by the node target declarations in M. With $\mathsf{shapes(M)}$ we refer to all the shape names in a document M.

Definition 3. *Given a graph G, and a* SHACL *document M, an* assignment σ *for G and M is a function mapping nodes in* $\mathsf{nodes(G, M)}$*, to subsets of* $\mathsf{shapes(M)} \cup \{\neg \mathtt{s} | \mathtt{s} \in \mathsf{shapes(M)}\}$*, such that for all nodes n and shape names $\mathtt{s}$, $\sigma(n)$ does not contain both $\mathtt{s}$ and $\neg\mathtt{s}$.*

Expression $[\![d]\!]^{n,G,\sigma}$ denotes the evaluation of constraint d on a node n w.r.t. a graph G under an assignment σ, as defined in [13]. If $[\![d]\!]^{n,G,\sigma}$ is True (resp. False) we say that node n satisfies (resp. does not satisfy) constraint d w.r.t. G under σ. For any graph G and assignment σ, fact $\mathtt{s} \in \sigma(n)$ (resp. $\neg\mathtt{s} \in \sigma(n)$) denotes the fact that node n conforms (resp. does not conform) to $\mathtt{s}$ w.r.t. G under σ. Expression $[\![d]\!]^{n,G,\sigma}$ evaluates to True, False or $\mathsf{Undefined}$ values of Kleene's 3-valued logic, and the truth value of any shape reference in d is computed using the assignment (it should be noted that the $\mathsf{Undefined}$ value never occurs in non-recursive shapes, but it is used to define possible extended semantics in the face of recursion). In other words, whenever a truth value in the evaluation of $[\![d]\!]^{n,G,\sigma}$ depends on whether another node j conforms to a shape $\mathtt{s}'$, with constraints d', this is not resolved by evaluating $[\![d']\!]^{j,G,\sigma}$, but instead it is True if $\mathtt{s}' \in \sigma(j)$, False if $\neg\mathtt{s}' \in \sigma(j)$, or else $\mathsf{Undefined}$. This, in turn, eliminates the problem of a potentially infinite series of constraint evaluations.

The semantics of SHACL validation can be defined with respect to a particular type of assignments, called *faithful* [13].

Definition 4. *For all graphs G,* SHACL *documents M and assignments σ, assignment σ is* faithful *w.r.t. G and M, denoted with $(G, \sigma) \models M$, if the following two conditions hold for any shape $\langle \mathtt{s}, t, d \rangle$ in* $\mathsf{shapes(M)}$ *and node n in* $\mathsf{nodes(G, M)}$ *:*

(1) $\mathtt{s} \in \sigma(n)$ iff $[\![d]\!]^{n,G,\sigma}$ is True*; and $\neg\mathtt{s} \in \sigma(n)$ iff $[\![d]\!]^{n,G,\sigma}$ is* False*;*
(2) if $G \models t(n)$ then $\mathtt{s} \in \sigma(n)$.

Condition (1) ensures that the facts denoted by the assignment are correct; while condition (2) ensures that the assignment is compatible with the target definitions. Condition (2) is trivially satisfied for SHACL documents where all target definitions are empty. Later we will want to discuss assignments where

the first property of Def. 4 holds, but not necessarily the second, in order to reason about the existence of alternative assignments that are correct (as in, they satisfy the first part of Def. 4) but that are not faithful. In fact, these will be faithful assignments to a document that is "stripped empty" of target definitions. Let $M^{\setminus t}$ denote the SHACL document obtained from substituting all target definitions in SHACL document M with the empty set. The following lemma holds:

Lemma 1. *For all graphs G,* SHACL *documents M and assignments σ, condition (1) from Definition 4 holds for any shape $\mathtt{s}$ in* $\mathsf{shapes(M)}$ *and node n in* $\mathsf{nodes(G, M)}$ *iff* $(G, \sigma) \models M^{\setminus t}$.

The existence of a faithful assignment is a necessary and sufficient condition for validation for non-recursive SHACL documents [13]. As we will see later, this is also necessary condition for all the other extended semantics.

Definition 5. *A graph G is valid w.r.t. a non-recursive* SHACL *document M if there exists an assignment σ such that $(G, \sigma) \models M$.*

5 SHACL Recursion

The semantics of recursion in SHACL documents is left undefined in the SHACL specification [23], and this gives rise to several possible interpretations. In this section we consider *extended* semantics of SHACL that define how to validate graphs against recursive SHACL documents. We focus on existing extended semantics that follow monotone reasoning. These can be characterised by two dimensions, namely the choice between *partial* and *total* assignments [13] and between *brave* and *cautious* validation [3], which we will subsequently define. Put together, these two dimensions define the four extended semantics of *brave-partial*, *brave-total*, *cautious-partial* and *cautious-total*. We will not go into the details of the less obvious dimension of *stable-model* semantics [3], which relates SHACL to non-monotone reasoning in logic programs.

As mentioned in the previous section, assignments can specify a truth value of True, False or $\mathsf{Undefined}$ to whether a node conforms to given shape. The truth value of $\mathsf{Undefined}$, which does not occur in non-recursive SHACL documents, can instead play an important role in validating SHACL under recursion. Intuitively, this happens during validation, when recursion makes it impossible for a node n to either conform or not to conform to a shape $\mathtt{s}$ but, at the same time, validity does not depend on whether n conforms to shape $\mathtt{s}$ or not. Consider for example the following SHACL document, containing a single shape $\langle \mathtt{s}^*, \emptyset, d^* \rangle$ (with name `:InconsistentS` in this example). This shape is defined as the negation of itself, that is, given a node n, a graph G and an assignment σ, fact $[\![d^*]\!]^{n,G,\sigma}$ is true iff $\neg\mathtt{s}^* \in \sigma(n)$, and false iff $\mathtt{s}^* \in \sigma(n)$.

```
:InconsistentS a sh:NodeShape ;
  sh:not :InconsistentS .
```

It is easy to see that any assignment that maps a node to either $\mathtt{s}^*$ or $\neg\mathtt{s}^*$ is not faithful, as it would violate condition (1) of Definition 4. However, an assignment that maps every node of a graph to the empty set would be faithful for that graph and document $\{\mathtt{s}^*\}$. Intuitively, this means that nodes in the graph cannot conform nor not conform to shape $\mathtt{s}^*$, but since this shape does not have any target node to validate, then the graph can still be valid. The fact of whether nodes conform or not conform to shape $\mathtt{s}^*$ can thus be left as "undefined".

This type of validation, for recursive SHACL documents, is called validation with partial assignments. More specifically, validation under brave-partial semantics simply extends the criterion of Def. 5 to recursive SHACL documents. All other extended semantics are constructed by adding additional conditions to brave-partial semantics. The term "partial" should not be interpreted as the fact that it describes only "part" of nodes of a graph, or that it describes the relationship of a node to only "part" of the shapes. Within a partial assignment, the conformance of every node to every shape is precisely specified by one of three truth values, and the term "partial" only indicates that one of these three truth values is Undefined.

Definition 6. *A graph G is valid w.r.t. a* SHACL *document M under brave-partial semantics if there exists an assignment σ such that $(G, \sigma) \models M$.*

In the SHACL specification, nodes either conform to, or not conform to a given shape, and the concept of an "undefined" level of conformance is arguably alien to the specification. It is natural, therefore, to consider restricting the evaluation of a constraint to the True and False values of boolean logic. This is achieved by restricting assignments to be *total*.

Definition 7. *An assignment σ is* total *w.r.t. a graph G and a* SHACL *document M if, for all nodes n in* $\mathsf{nodes(G, M)}$ *and shapes $\langle \mathtt{s}, t, d\rangle$ in M, either $\mathtt{s} \in \sigma(n)$ or $\neg\mathtt{s} \in \sigma(n)$.*

For any graph G and SHACL document M we denote with $A^{G,M}$ and $A_T^{G,M}$, respectively, the set of assignments, and the set of total assignments for G and M. Trivially, $A_T^{G,M} \subseteq A^{G,M}$ holds.

Definition 8. *A graph G is valid w.r.t. a* SHACL *document M under brave-total semantics if there exists an assignment σ in $A_T^{G,M}$ such that $(G, \sigma) \models M$.*

Since total assignments are a more specific type of assignments, if a graph G is valid w.r.t. a SHACL document M under brave-total semantics, than it is also valid w.r.t. M under brave-partial semantics. The reverse, instead, is only true for non-recursive SHACL documents. In fact, as shown in [13], if there exists a faithful assignment for a graph G and a non-recursive document M, then there exists also a total faithful assignment for G and M. Therefore, the definition of validity under brave-total semantics (Def. 8), for non-recursive SHACL documents, coincides with the standard definition of validation (Def. 5).

While total assignments can be seen as a more natural way of interpreting the SHACL specification, they are not without issues when recursive SHACL documents are considered. Going back to our previous example, we can notice that there cannot exist a total faithful assignment for the SHACL document containing shape :`InconsistentS`, for any non-empty graph. This is a trivial consequence of the fact that no node can conform to, nor not conform to, shape :`InconsistentS`. This, however, is in contradiction with the SHACL specification, which implies that a SHACL document without target declarations in any of its shapes (such as the one in our example) should trivially validate any graph. If there are no target declarations, in fact, there are no target nodes on which to verify the conformance of certain shapes, and thus no violations should be detected.

The second and last dimension that we consider is the difference between brave and cautious validation. When a SHACL document M is recursive, there might exist multiple assignments σ satisfying property (1) of definition 4, that is, such that $(G, \sigma) \models M^{\setminus t}$. Intuitively, these can be seen as equally "correct" assignments with respect to the constraints of the shapes, and brave validation only checks whether at least one of them is compatible with the target definitions of the shapes. Cautious validation, instead, represents a stronger form of validation, where all such assignments must be compatible with the target definitions.

Definition 9. *A graph G is valid w.r.t. a* SHACL *document M under* cautious-partial *(resp.* cautious-total*) semantics if it is (1) valid under* brave-partial *(resp.* brave-total*) semantics and (2) for all assignments σ in $A^{G,M}$ (resp. $A_T^{G,M}$), it is true that if $(G, \sigma) \models M^{\setminus t}$ holds then $(G, \sigma) \models M$ also holds.*

To exemplify this distinction, consider the following SHACL document M_1. This document requires the daily special of a restaurant, node :`DailySpecial`, to be vegetarian, that is, to conform to shape :`VegDishShape`. This shape is recursively defined as follows. Something is a vegetarian dish if it contains an ingredient, and all of its ingredients are vegetarian, that is, entities conforming to the :`VegIngredientShape`. A vegetarian ingredient, in turn, is an ingredient of at least one vegetarian dish.

```
:VegDishShape a sh:PropertyShape ;
   sh:targetNode :DailySpecial ;
   sh:path :hasIngredient ;
   sh:minCount 1 ;
   sh:qualifiedMaxCount 0 ;
   sh:qualifiedValueShape [ sh: not :VegIngredientShape ] .

:VegIngredientShape a sh:PropertyShape ;
   sh:path [ sh:inversePath :hasIngredient ] ;
   sh:node :VegDishShape .
```

Consider now a graph G_1 containing the following triple.

```
:DailySpecial :hasIngredient :Chicken .
```

Due to the recursive definition of :VegDishShape, there exist two different assignments σ_1 and σ_2, which are both faithful for G_1 and $M_1^{\setminus t}$. In σ_1, no node in G_1 conforms to any shape, while σ_2 differs from σ_1 in that node :DailySpecial conforms to :VegDishShape and node :Chicken conforms to :VegIngredientShape. Essentially, either both the dish and the ingredient from graph G_1 are vegetarian, or neither is. Therefore, σ_2 is faithful for G_1 and M_1, while σ_1 is not. The question of whether the daily special is a vegetarian dish or not can be approached with different levels of "caution". Under brave validation, graph G_1 is valid w.r.t. M_1, since it is possible that the daily special is vegetarian. Cautious validation, instead, takes the more conservative approach, and under its definition G_1 is not valid w.r.t. M_1, since it is also possible that the daily special is not vegetarian. When analysing such recursive definitions, one might want to exclude "unfounded" assignments, that is, assignments that assign certain shapes to a node for no other reason than to allow the validation of a graph. This is achieved by the recursive semantics for SHACL proposed in [3], which is based on the concept of *stable models* from Answer Set Programming.

For each extended semantics, the definition of validity of a graph G with respect to a SHACL document M, denoted by $G \models M$, is summarised in the following list.

brave-partial there exists an assignment that is faithful w.r.t. G and M;
brave-total there exists an assignment that is total and faithful w.r.t. G and M;
cautious-partial there exists an assignment that is faithful w.r.t. G and M, and every assignment that is faithful w.r.t. G and $M^{\setminus t}$ is also faithful w.r.t. G and M.
cautious-total there exists an assignment that is total and faithful w.r.t. G and M, and every assignment that is total and faithful w.r.t. G and $M^{\setminus t}$ is also faithful w.r.t. G and M.

6 Formal languages for SHACL

In this section we review the two main formal languages that have been proposed to model the semantics of SHACL. We first discuss a complete first-order formalisation of SHACL, which can be used to study a number of decision problems. We then present a simplified language that effectively models SHACL constraints for the purpose of validation.

6.1 SCL, A First-Order Language for SHACL

In order to formally study SHACL, it is convenient to abstract away from the syntax of its RDF and SPARQL representations. The SCL first order language [37,36] is currently the only complete formalisation of SHACL into a formal logical system. The expressiveness of this language covers all of the SHACL target declarations and all of the SHACL core constraint components, including the filter components, which are less commonly studied. This language captures the semantics

of whole SHACL documents, and it can be used to study a number of related decision problems, including validation. The relation between SHACL and SCL is given by translation τ [36], such that, given a SHACL document M, the first order sentence $\tau(M)$ is the translation of M into SCL. We identify the inverse translation with τ^-.

Before defining SCL and its properties, we must define how RDF graphs and assignments are modelled in this logical framework. The domain of discourse is assumed to be the set of RDF terms. Triples are modelled as binary relations, with atom $R(s, o)$ corresponding to triple $<s, R, o>$. A minus sign identifies the *inverse* role, i.e. $R^-(s, o) = R(o, s)$. Binary relation name isA represents class membership triples $<s, \texttt{rdf:type}, o>$ as $\texttt{isA}(s, o)$. Assignments are modelled with a set of monadic relations Σ, called *shape relations*. Each SHACL shape s is associated with a unique shape relation $\Sigma_{\texttt{s}}$ in SCL. Facts $\Sigma(x)$ (resp. $\neg\Sigma(x)$) describe an assignment σ such that $\texttt{s} \in \sigma(x)$ (resp. $\neg\texttt{s} \in \sigma(x)$). Since this logical framework adopts boolean logic, $\forall x.\ \Sigma(x) \vee \neg\Sigma(x)$ holds, by the law of excluded middle. Thus shape relations define total assignments.

Given a graph G and an assignment σ, we now define their respective translations G^τ and σ^τ into first order structures.

Definition 10. *Given a graph G, fact $p(s, o)$ is true in the first order structure G^τ iff $<s, p, o> \in G$.*

Definition 11. *Given a total assignment σ, fact $\Sigma_{\texttt{s}}(n)$ is true in the first order structure σ^τ iff $\texttt{s} \in \sigma(n)$.*

Definition 12. *Given a graph G and a total assignment σ, the first order structure I* induced *by G and σ is the disjoint union of structures G^τ and σ^τ. Given a first order structure I: (1) the graph G* induced *by I is the graph that contains triple $<s, p, o>$ iff $I \models p(s, o)$ and (2) the assignment σ induced by I is the assignment such that, for all nodes n and shape relations $\Sigma_{\texttt{s}}$, fact $\texttt{s} \in \sigma(n)$ is true iff $I \models \Sigma_{\texttt{s}}(n)$ and $\neg\texttt{s} \in \sigma(n)$ iff $I \not\models \Sigma_{\texttt{s}}(n)$.*

The existence of faithful assignments using SCL and its standard model-theoretic semantics is presented in the following theorem [37]. Trivially, this also defines what condition, in SCL, corresponds to validation under the brave-total extended semantics (Def. 8), which also defines validation for non-recursive SHACL documents (Def. 5).

Theorem 1. *For any graph G, total assignment σ and* SHACL *document M, it is true that $(G, \sigma) \models M$ iff $I \models \tau(M)$, where I is the first order structure induced by G and σ.*

For any first order structure I and SCL formula ϕ, it is true $I \models \phi$ iff $(G, \sigma) \models \tau^-(\phi)$, where G and σ are, respectively, the graph and assignment induced by I.

Sentences in the SCL language follow the φ grammar in Definition 13.

Definition 13. *The* SHACL *first order language (SCL, for short) is the set of first order* sentences *built according to the following context-free grammar, where*

Table 2. Translation of a shape with name `s` with a target definition t, into an SCL target axiom.

Target declaration in t	SCL target axiom
Node target (node `c`)	$\Sigma_{\mathtt{s}}(\mathtt{c})$
Class target (class `c`)	$\forall x.\mathtt{isA}(x,\mathtt{c}) \rightarrow \Sigma_{\mathtt{s}}(x)$
Subjects-of target (relation R)	$\forall x, y.R(x, y) \rightarrow \Sigma_{\mathtt{s}}(x)$
Objects-of target (relation R)	$\forall x, y.R^{-}(x, y) \rightarrow \Sigma_{\mathtt{s}}(x)$

`c` is a constant from the domain of RDF *terms, Σ is a shape relation, F is a filter relation, with shape relations disjoint from filter relations, R is a binary-relation name, * indicates the transitive closure of the relation induced by $\pi(x, y)$, the superscript $\pm$ refers to a relation or its inverse, and $n \in \mathbb{N}$.*

$$
\begin{aligned}
\varphi &:= \top \mid \varphi \wedge \varphi \\
&\quad \mid \Sigma(\mathtt{c}) \mid \forall x\,.\,\mathtt{isA}(x, \mathtt{c}) \rightarrow \Sigma(x) \mid \forall x, y\,.\,R^{\pm}(x, y) \rightarrow \Sigma(x) \\
&\quad \mid \forall x.\ \Sigma(x) \leftrightarrow \psi(x); \\
\psi(x) &:= \top \mid \neg\psi(x) \mid \psi(x) \wedge \psi(x) \mid x = \mathtt{c} \mid F(x) \mid \Sigma(x) \mid \exists y.\,\pi(x, y) \wedge \psi(y) \\
&\quad \mid \neg\exists y.\,\pi(x, y) \wedge R(x, y) && \texttt{[D]} \\
&\quad \mid \forall y.\,\pi(x, y) \leftrightarrow R(x, y) && \texttt{[E]} \\
&\quad \mid \forall y, z\,.\,\pi(x, y) \wedge R(x, z) \rightarrow \varsigma(y, z) && \texttt{[O]} \\
&\quad \mid \exists^{\geq n} y\,.\,\pi(x, y) \wedge \psi(y); && \texttt{[C]} \\
\pi(x, y) &:= R^{\pm}(x, y) \\
&\quad \mid \exists z\,.\,\pi(x, z) \wedge \pi(z, y) && \texttt{[S]} \\
&\quad \mid x = y \vee \pi(x, y) && \texttt{[Z]} \\
&\quad \mid \pi(x, y) \vee \pi(x, y) && \texttt{[A]} \\
&\quad \mid (\pi(x, y))^{\star}; && \texttt{[T]} \\
\varsigma(x, y) &:= x <^{\pm} y \mid x \leq^{\pm} y.
\end{aligned}
$$

Symbol φ corresponds to a SHACL document. An SCL sentence could be empty ($\top$), a conjunction of documents, a *target axiom* representing a target definition (a production of the 3rd, 4th and 5th production rule) or a *constraint axiom* representing a constraint (a production of the last production rule). Target axioms take one of three forms, based on the type of target declarations. The translation of SHACL target declarations into SCL target axioms is summarised in Table 2. Letters in square brackets are annotations for naming SCL components and thus are not part of the grammar. These letters are essentially first-letter abbreviations of *prominent* SHACL components (that together define fragments of SCL), and are also listed in Table 3.

The non terminal symbol $\psi(x)$ corresponds to the subgrammar of the SHACL constraints components. Within this subgrammar, $\top$ identifies an empty constraint, $x = \mathtt{c}$ a constant equivalence constraint and F a monadic filter relation

Table 3. Relation between prominent SHACL components and SCL expressions.

Abbr.	Name	SHACL component	Corresponding expression
S	Sequence Paths	Sequence Paths	$\exists z \,.\, \pi(x,z) \wedge \pi(z,y)$
Z	Zero-or-one Paths	sh:zeroOrOnePath	$x = y \vee \pi(x,y)$
A	Alternative Paths	sh:alternativePath	$\pi(x,y) \vee \pi(x,y)$
T	Transitive Paths	sh:zeroOrMorePath sh:oneOrMorePath	$(\pi(x,y))^\star$
D	Property Pair Disjointness	sh:disjoint	$\neg\exists y.\pi(x,y) \wedge R(x,y)$
E	Property Pair Equality	sh:equals	$\forall y \,.\, \pi(x,y) \leftrightarrow R(x,y)$
O	Property Pair Order	sh:lessThan sh:lessThanOrEquals	$x \leq^{\pm} y$ and $x <^{\pm} y$
C	Cardinality Constraints	sh:qualifiedValueShape sh:qualifiedMinCount sh:qualifiedMaxCount	$\exists^{\geq n} y \,.\, \pi(x,y) \wedge \psi(y)$ with $n \neq 1$

(e.g. $F^{\text{IRI}}(x)$, true iff x is an IRI). Filters components are captured by $F(x)$ and the O component. The C component captures qualified value shape cardinality constraints. The E, D and O components capture the equality, disjointedness and order property pair components. The $\pi(x,y)$ subgrammar models SHACL property paths. Within this subgrammar S denotes sequence paths, A denotes alternate paths, Z denotes a zero-or-one path and T denotes a zero-or-more path.

Translation τ results in a subset of SCL formulas, called *well-formed* defined subsequently, and the inverse translation τ^- only takes well formed sentences as an input. An SCL formula ϕ is well-formed iff for every shape relation Σ, formula ϕ contains exactly one constraint axiom with relation Σ on the left-hand side of the implication. Intuitively, this condition ensures that every shape relation is "defined" by a corresponding constraint axiom. The translation of the document from Fig. 2, into a well-formed SCL sentence, via τ, is the following. Arguably, this logic notation might seem easier to read and understand than the SHACL syntax of Fig. 2.

$$\Big(\forall x.\ \texttt{isA}(x, \texttt{:Employee}) \rightarrow \Sigma_{\texttt{:EmployeeShapeB}}(x)\Big)$$
$$\wedge \Big(\forall x.\ \Sigma_{\texttt{:EmployeeShapeB}}(x) \leftrightarrow \exists y.\ R_{\texttt{:hasOfficeNumber}}(x,y) \wedge \Sigma_{\texttt{:OfficeNumberShape}}(y)\Big)$$
$$\wedge \Big(\forall x.\ \Sigma_{\texttt{:OfficeNumberShape}}(x) \leftrightarrow F^{\text{length}\geq 3}(y)\Big)$$

The language defined without any of these constructs is called the *base* language, denoted $\varnothing$. On top of the base language different syntactic fragments of SCL are defined by considering different combinations of features allowed. We name these fragments by concatenating the letters that represent the features allowed, into a single name. For example, SA identifies the fragment that only allows the base language, sequence paths and alternate paths. This means that in order to write an SCL document in SA, one can only use the production rules

of Def. 13 that are not annotated with any feature (base language) or those identified by abbreviations `S` and `A`.

The SHACL specification presents an unusual asymmetry in the fact that equality, disjointedness and order components force one of their two path expressions to be an atomic relation. This can result in situations where the order constraints can be defined in just one direction, since only the less-than and less-than-or-equal property pair constraints are defined in SHACL. The `O` fragment models a more natural order comparison that includes the $>$ and $\geq$ components. The fragment where the order relations in the $\varsigma(x, y)$ subgrammar cannot be inverted is denoted `O'`.

When interpreting an `SCL` sentence, particular care should be paid to the semantics of filter relation. The interpretation of each filter relation, such as $F^{\text{IRI}}(x)$, is the subset of the domain of discourse on which the filter is true. This interpretation is constant across all models, and defines the semantics of the filter. When considering the decision problem of validation, filter relations in `SCL` must be suitably defined by interpreted relations (similarly to how the equality operator is). When considering additional decision problems, such as satisfiability and containment (which will be discussed in Section 7), the semantics of filters can be axiomatisated, thus removing the need for special interpreted relations. The filter axiomatisation presented in [37] captures the semantics of all SHACL filters with the single exception of `sh:pattern`, as this filter defines complex non-standard regular expressions based on the SPARQL REGEX function [42].

6.2 $\mathcal{L}$, a Language for SHACL Constraint Validation

Another major language used to study SHACL is $\mathcal{L}$ which was presented in [13] and paved the way to subsequent formal studies of SHACL. The $\mathcal{L}$ language differs from `SCL` in scope and purpose. While `SCL` sentences describe whole SHACL documents, sentences in $\mathcal{L}$ describe individual SHACL constraints. The $\mathcal{L}$ language is primarily designed to investigate the complexity of SHACL validation. As such, it relies on assumptions that do not hold when studying other decision problems such as satisfiability and containment, which, instead, can be studied using `SCL`. In particular, $\mathcal{L}$ assumes that all filter components can be evaluated on a node in constant time, and thus are all equivalent, for the purposes of validation. Thanks to this reduced scope, $\mathcal{L}$ seems less complex than `SCL`, and it is a useful formalism to study the evaluation of SHACL constraints. The semantics of an $\mathcal{L}$ sentence ϕ is defined in [13] through the use of faithful assignments. In particular, [13] fixes a lookup table that provides the truth value of the evaluation of ϕ on a node n for a graph G and an assignment σ. Instead, `SCL` relies on the standard model-theoretic semantics.

The grammar of $\mathcal{L}$ sentences is given next. In this grammar s is a shape name; I is an IRI; r is a SHACL property path; n is a positive integer.

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

Table 4 defines the correspondence between $\mathcal{L}$ and the $\psi(x)$ sub-grammar of `SCL`. It is easy to see that $\mathcal{L}$ sentences correspond to a subset of the $\psi(x)$ sub-

Table 4. Correspondence between an $\mathcal{L}$ sentence ϕ, and SCL $\psi^{\phi}(x)$ expressions, such that a constraint ϕ is satisfied on a node n w.r.t. a graph G and an assignment σ iff $I \models \psi^{\phi}(n)$, where I is the first order structure induced by G and σ. It is assumed that paths are expressed using the $\pi(x, y)$ subgrammar of SCL, and that r_2 is an IRI.

$\mathcal{L}$ expression ϕ	Corresponding SCL $\psi^{\phi}(x)$
$\top$	$\top$
s	$\Sigma_{\mathtt{s}}(x)$
I	$x = I$
$\phi_1 \wedge \phi_2$	$\psi^{\phi_1}(x) \wedge \psi^{\phi_2}(x)$
$\neg\phi$	$\neg\psi^{\phi}(x)$
$\geq_n r.\phi$	$\exists^{\geq n} y\,.\, r(x, y) \wedge \psi^{\phi}(y)$
$\mathrm{EQ}\,(r_1, r_2)$	$\forall y.\, r_1(x, y) \leftrightarrow r_2(x, y)$

grammar of SCL, assuming that r_2 denotes a predicate path. This assumption is required as in $\mathcal{L}$ both arguments of $\mathrm{EQ}\,(r_1, r_2)$, which captures the SHACL equality operator (`sh:equals`), are path expressions. This is a generalisation of SHACL, since the SHACL specification requires one of the two paths to be a simple predicate path, or in other words, an IRI. It should also be noted that $\mathcal{L}$ does not model property pair order components (denoted `O` in SCL), and that the `sh:closed` component is modelled using path expression operators not supported by SHACL paths. The SHACL disjoint constraint component (denoted `D` in SCL) is only implicitly included in $\mathcal{L}$ when considering recursion. It is possible, in fact, to represent a disjoint constraint component in $\mathcal{L}$ using two auxiliary recursive shapes [13].

7 SHACL Decision Problems

Several existing pieces of work in the literature focus on SHACL, and several related decision problems have been investigated. In Section 7.1 we review existing work on the core decision problem for SHACL, namely validation. Unlike validation, which studies the relationship between a SHACL document and an RDF graph, the decision problems of *satisfiability* and *containment*, reviewed in Section 7.2, focus on intrinsic properties of SHACL documents and their components.

7.1 Validation

Validation is a core decision problem for SHACL, since the main application of this language is the validation of RDF graphs. This decision problem is decidable for all of the semantics discussed in this article, including the four extended semantics. The complexity lower bounds for validation, however, depend on the fragment of SHACL being considered. Table 5 lists the data complexity of three fragments of SHACL given in [13,3]. The three fragments are (1) $\text{SHACL}^{non-rec}$,

Table 5. Data complexity of SHACL validation, results from [12].

Fragment	Data complexity of validation
SHACL$^{non-rec}$	NL-c
SHACL^{+}	PTIME-c
SHACL^{rec}	NP-c

the fragment of non-recursive SHACL documents built using $\mathcal{L}$ constraints; (2) SHACL^{+}, the fragment of SHACL documents built using $\mathcal{L}$ constraints with a restricted use of negation, that is, substituting the $\neg\phi$ production rule of $\mathcal{L}$ into $\phi_1 \vee \phi_2$; and (3) SHACL^{rec}, the fragment of SHACL documents built using $\mathcal{L}$ constraints. The most expressive of these fragments, SHACL^{rec}, is NP-complete in data complexity.

7.2 Satisfiability and Containment

Satisfiability and containment are standard decision problems that have been investigated in the context of SHACL. These two decision problems, unlike validation, do not take a graph as an input. Instead, they focus on SHACL documents, shapes or constraints. Given any notion of validity from one of the semantics defined earlier, the following decision problems are defined. For simplicity, when discussing satistiability and containment, we will assume the use of the semantics of validation from Definitions 8 and 5.

Definition 14. *A* SHACL *document M is* satisfiable *iff there exists a graph G such that $G \models M$. Deciding whether a* SHACL *document is satisfiable is the decision problem of* SHACL satisfiability.

Definition 15. **SHACL *Containment*:** *For all* SHACL *documents M_1, M_2, we say that M_1 is* contained *in M_2, denoted $M_1 \subseteq M_2$, iff for all graphs G, if $G \models M_1$ then $G \models M_2$. Deciding whether a* SHACL *document is contained in another is the decision problem of* SHACL containment.

Two SHACL documents M_1 and M_2 that are contained in each other ($M_1 \subseteq M_2$ and $M_2 \subseteq M_1$) are *semantically equivalent*. Two semantically equivalent documents are not necessarily equivalent syntactically, since in SHACL the same constraint can be expressed using different sets of shapes.

The satisfiability and containment decision problems for SHACL can be polynomially reduced to the satisfiability decision problem for SCL, defined as follows in the natural way [37].

Definition 16. *An SCL sentence ϕ is satisfiable iff there exists structure Ω such that $\Omega \models \phi$. Deciding whether a SCL sentence is satisfiable is the decision problem of SCL* satisfiability.

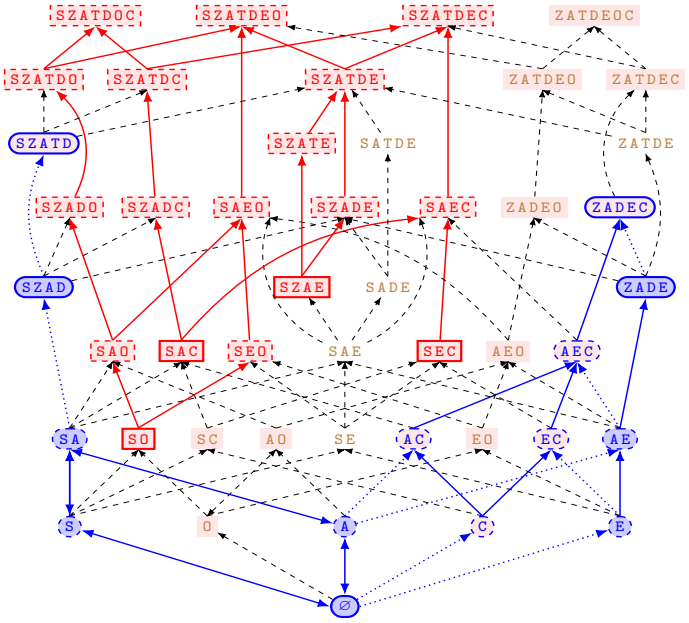

Fig. 3. [37] Decidability and complexity map of SCL satisfiability. Round (blue) and square (red) nodes denote decidable and undecidable fragments, respectively. Solid borders on nodes correspond to theorems in this paper, while dashed borders are implied results. Directed edges indicate inclusion of fragments, while bidirectional edges denote polynomial-time reducibility. Solid edges are preferred derivations to obtain tight results, while dotted ones leads to worst upper-bounds or model-theoretic properties. Finally, a light blue background indicates that the fragment enjoys the finite-model property, while those with a light red background do not satisfy this property.

This reduction allows us to study the decidability and complexity of the SHACL satisfiability and containment problems for a given SHACL fragment by studying the decidability and complexity of SCL satisfiability, for the corresponding fragments. The results of this study, published in [37], are summarised in Figure 3. Negative results indicate the undecidability of both the SCL fragment, and the corresponding SHACL fragment. Positive results, shown in round blue in the figure, indicate that both satisfiability and containment are decidable, for that fragment of non-recursive SHACL, and are accompanied with complexity upper-bounds. Starting from the negative results, SHACL satisfiability and containment is, in general, undecidable. This was shown even for several non-recursive fragments, through a semi-conservative reduction from the standard domino problem [50,5,39], which is an undecidable decision problem. More specifically, the SHACL satisfiability problems for the SO, SAC, SEC, SEO', and SZAE fragments are undecidable [37].

Positive results are obtained by noticing that several SCL fragments are included in decidable fragments of first order logic. For example, the SZATD fragment of SCL is included in the extension of the unary-negation fragment of first-order logic with arbitrary transitive relations, which can be solved in 2ExpTime [1,20,15]. The complexity upper bounds identified in [37] for SHACL fragments range from ExpTime to 2ExpTime. Both decision problems are defined over SHACL documents which, similarly to the schema of a dataset, could be assumed to be of small or constant size.

Up until this point we considered the satisfiability and containment problems defined at the level of SHACL documents. However, it is possible to study variations of these problems at different levels of granularity. For example, the satisfiability and containment problems at the level of SHACL constraints are defined in [37], and are shown to be reducible to the problem of SHACL satisfiability. An approach that uses Description Logics Reasoning is presented in [26] to compute *shape containment*, that is, containment at the level of shapes, for a restrictive fragment of SHACL, which however allows recursion.

8 Inference Rules and the Schema Expansion

Datasets are often dynamic objects, which are frequently subject to modification. When an RDF graph is modified, its validity w.r.t. a SHACL document might change. If the modifications that a dataset undergoes are completely arbitrary, then it is not possible to make predictions regarding validity, and the dataset might need to be re-validated after each modification. Many types of modifications that can be applied on a dataset, however, are predictable or a result of some reasoning process. In particular, many types of modifications can be represented as *inference rules* $B \to H$, where a set of facts H, called the *head* are added to a dataset whenever a query B, called the *body*, finds a match on the dataset. Given an RDF graph G, and a set of inference rules R, it is possible to compute graph G', *closure* of G under R, by applying the *chase* algorithm [4]. The chase algorithm, intuitively, consists in repeatedly applying the rules of R on G until convergence. For simplicity, we assume that the chase algorithm is guaranteed to terminate for the inference rules considered.

Assuming graph G is valid w.r.t. to a SHACL document M, the approach presented in [38], called *schema expansion*, allows us to predict whether the graph closure G' will still be valid w.r.t. M without having to validate G' against M. In particular, given a SHACL document M and a set of inference rules R, the schema expansion process computes the "maximal sub-document" of M which will still validate G after the rule applications. That is, the schema expansion is a SHACL document M', called *schema consequence*, such that (1) $M \subseteq M'$ (i.e., M' is a subset of the restrictions of M); (2) validity is preserved after closure, that is, for any graph G valid w.r.t. M, its closure G' under R is valid w.r.t. M'; and (3) M' is "minimally-containing", i.e., there is no document M'' that satisfies conditions (1) and (2) and such that that $M'' \subset M'$. If a schema consequence M' of a SHACL document M under inference rules R is semantically equivalent

to M, then any graph G, valid w.r.t. M is guaranteed to remain valid w.r.t. M after computing its closure under R. In other words, this means that the application of rules R cannot "invalidate" graphs valid w.r.t. document M.

Consider, for example, the following graph G_1, which describes :Eve, a manager of the company in the IT department, and one of her subordinates :Fiona.

```
:Eve a :Manager ;
    :hasDepartment "IT" .
:Fiona a :Employee ;
    :hasManager :Eve .
```

This graph is valid w.r.t. the following SHACL document M_1, which states that each employee must have a manager, and each manager must have a department.

```
:SubordinateS a :PropertyShape ;
    sh:targetClass :Employee ;
    sh:path :hasManager ;
    sh:minCount 1 .

:ManagerS a sh:PropertyShape ;
    sh:targetClass :Manager ;
    sh:path :hasDepartment ;
    sh:minCount 1 .
```

Consider now the set of inference rules $R_1 = \{r_1, r_2\}$, where rules r_1 and r_2 are defined as follows. For simplicity, we represent both the head and the body of rules as SPARQL graph patterns, which are interpreted as SPARQL CONSTRUCT queries where the WHERE and CONSTRUCT clauses are the body and the head, respectively. Rule r_1 states that every manager can be inferred to be an employee, and r_2 states that everyone can be inferred to be in the same department as their manager.

r_1 = `{?x rdf:type :Manager}` $\rightarrow$ `{?x rdf:type :Employee}`
r_2 = `{?x :hasManager ?y . ?y :hasDepartment ?z}` $\rightarrow$ `{?x :hasDepartment ?z}`

The closure of graph G_1 under rules R_1 is the following graph G_2.

```
:Eve a :Manager ;
    a :Employee ;
    :hasDepartment "IT" .
:Fiona a :Employee ;
    :hasManager :Eve ;
    :hasDepartment "IT" .
```

Notice that graph G_2 is not valid w.r.t. M_1, since :Eve violates :SubordinateS, but it is valid w.r.t. another document M_2 which only contains shape :ManagerS. In fact, M_2 is a schema consequence of M_1 and R_1. Therefore, we know that the closure under R_1 of any graph valid w.r.t. M_1 will validate shape :ManagerS, but it might not validate :SubordinateS.

Two approaches to compute the schema expansion are presented in [38], for datalog [9] inference rules without negation. The first based on the concept

of *critical instance* [30], and the second an optimisation of the first. These approaches are only defined on a fragment of SHACL that, although restricted, is sufficient to express common constraints for RDF validation, such as the Data Quality Test Patterns TYPEDEP, TYPRODEP, PVT, RDFS-DOMAIN and RDFS-RANGE in the categorisation by Kontokostas et al. [24]. Intuitively, the difficulty in computing a schema expansion lies in having to consider all possible graphs that are valid w.r.t. a SHACL document, and their interactions with arbitrarily complex inference rules.

9 Applications, Tools and Implementations

Over a few years since reaching its status as a W3C recommendation, the level of maturity and adoption of the SHACL technology has been steadily increasing. In this section we review existing SHACL implementations, tools designed to facilitate the creation and management of SHACL documents, and documented usages of SHACL in practical applications.

9.1 Tools for SHACL Validation

The availability of mature tools is often a crucial requirement for the widespread adoption of a technology. To date, SHACL validation has been integrated in a number of mainstream tools and triplestores.[4] An example of this is RDF4J,[5] a Java framework for managing RDF data, which now includes an engine for SHACL validation. The RDF4J framework is integrated in a number of projects, most notably the GraphDB[6] triplestore. Other SHACL-enabled databases include AllegroGraph[7] by Franz Inc, Apache Jena[8] by Apache, and Stardog[9] by Stardog Union Inc. A benchmark for the comparison of different SHACL implementation was proposed in [41], along with results for four different databases. A SHACL implementation is also available for Python through the pySHACL[10] library.

One of the first tools to enable the validation of recursive SHACL graphs was SHACL2SPARQL [11]. Another tool, Trav-SHACL [18], implements a SHACL engine designed to optimise the evaluation of SHACL core constraint components expressible in fragments of the $\mathcal{L}$ language [13]. On these fragments of SHACL, Trav-SHACL was shown to achieve significantly faster validation times compared to the SHACL2SPARQL tool.

[4] `https://w3c.github.io/data-shapes/data-shapes-test-suite/` accessed on 18/6/21
[5] `https://rdf4j.org/` accessed on 18/6/21
[6] `https://graphdb.ontotext.com/` accessed on 18/6/21
[7] `https://allegrograph.com/` accessed on 18/6/21
[8] `https://jena.apache.org/` accessed on 18/6/21
[9] `https://www.stardog.com/` accessed on 18/6/21
[10] `https://pypi.org/project/pyshacl/` accessed on 18/6/21

9.2 Tools for Generating SHACL Documents

While efficient tools to perform graph validation are undoubtedly essential to the widespread adoption of SHACL, it is also important to devise practical ways to generate suitable SHACL documents, without which validation would not be possible. On the one hand, SHACL documents can be manually created by experts. Tools to support this manual process can facilitate this, especially when integrated with already established software. An example of this is SHACL4P [17], a plugin for the Protégé ontology editor [31] which includes an editor to create SHACL documents, and a validator that allows users to test the document by validating an ontology with it, and then visualising any constraint violations. Shape Designer [6] is another tool to create SHACL documents that combines a graphical editor, and additional algorithms to create constraints semi-automatically by analysing the data graph. The benefits of different types of visualisations as an aid to the creating and editing of constraints for RDF graphs was studied in [28]. Existing work also investigated the possibility of generating SHACL documents from natural language text [43].

A number of approaches have been designed to automate the creation of SHACL documents. The SHACLearner [33] approach, generates SHACL documents by learning a kind of rules called Inverse Open Path (IOP) Rules from the graph data provided. IOP rules are strongly related to `SCL` and therefore SHACL. An IOP rule essentially follows the same structure as an `SCL` constraint axiom, both syntactically and semantically, with the only exception that the iff operator is replaced by a rightward implication. Another approach to automate the creation of SHACL documents is the Astrea-KG Mappings [10]. These mappings consist of a set of manually created mappings from OWL [49] to SHACL, that can be used to automatically generate SHACL documents from OWL ontologies. As described in [34], SHACL documents can also be generated from the axioms defined by ontology design patterns [19]. The approach from [44] generates SHACL documents for the purpose of quality assessment, using the ontology design patterns and data statistics created by the ABSTAT [45] tool as an input. Another similar approach, presented in [8], allows the automatic extraction of SHACL constraints from a SPARQL endpoint, and was tested on the dataset of Europeana[11].

Notably, SHACL documents can also be seen as describing a desirable "schema" for graph data. As such, they can be used as a template to generate new RDF data. An example of this is the Schímatos [51] tool, which generates forms for RDF graph editing based on SHACL documents, in order to simplify the graph editing task, and minimize the chance of error.

9.3 Adoption of SHACL

An analysis of existing use cases of SHACL can be useful to gain insights on how this technology is used in practice, and on in its level of adoption. In a recent review, 13 existing projects using SHACL have been reviewed, and the most common constraints observed were cardinality, class, datatype and disjunction [29].

[11] `https://www.europeana.eu/en` accessed on 18/6/21

Several works investigate the use of SHACL to verify compliance of a dataset w.r.t. certain policies, such as GDPR requirements [35,2]. Other applications of SHACL include type checking program code [27] and detecting metadata errors in clinical studies [22]. SHACL is also used by the European Commission to facilitate data sharing, for example by validating metadata about public services against the recommended vocabularies [46]. Notably, several approaches define translations into SHACL from other technologies, such as ontologies and other schema and constraint languages [16,21,25,32,40,47]. These results show that SHACL tools, and in particular validators, can benefit areas where technologies other than SHACL are already established.

10 Conclusion

Within this review we examined SHACL, a constraint language that can be used to validate RDF graphs. These constraints can be used to describe the properties of a graph, to detect possible errors in the data or provide data quality assurances. In this review we first presented the main concepts of the SHACL specification, such as the concept of *shapes*, and their two main components, *targets* and *constraints*. We discussed the primary way to perform SHACL validation, using SPARQL queries, and how the semantics of validation can be abstracted with the concept of *assignments*.

While the SHACL specification describes how validation should be performed, its semantics is left implicit and not formally defined. We have extensively discussed studies that address this problem. In particular, we reviewed a complete formalisation of SHACL into a fragment of first order logic called `SCL`. This formalisation lays bare several properties of SHACL, and provides decidability and complexity results for several SHACL-related decision problems. Another important line of work focuses on defining potential extensions of SHACL semantics that can be used in the face of *recursion*. The SHACL specification, in fact, allows constraints to be recursively defined, but it does not define its semantics. We also presented existing work studying the interaction of SHACL with inference rules. Datasets are often dynamic objects, and several questions arise when considering the effects of this dynamism on the constraints imposed over them.

From the point of view of maturity and level of adoption of the SHACL technology, we reviewed several implementations of SHACL validators, which are now integrated in many mainstream RDF databases, and several tools designed to facilitate the creation and management of SHACL documents. Several approaches, in particular, provide automated or semi-automated ways of generating suitable SHACL documents from a diverse range of sources, such as graph data, ontologies, or natural language texts. Existing efforts in mapping other constraint/validation languages into SHACL is also worth noting, as it suggests that the usefulness of SHACL could be extended to support other existing technologies. While the true extent of SHACL adoption is hard to establish, since not all usages of SHACL are publicly documented, we found evidence of its usage in several areas, such

as to facilitate data sharing, to validate dataset against policies, and to detect errors in datasets.

Despite the wealth of work on this topic, SHACL is still a recent specification, and a number of important directions for future work still exist. For example, there are opportunities to optimise SHACL validators for particular type of constraints, or for particular scenarios, like for highly dynamic databases. More studies are needed to properly assess the usage of SHACL in practical applications, and what types of constraints are more commonly used and how. While the full semantics of SHACL has been formally defined, more work is needed to formally establish its relation with other constraint languages. It is also important to notice that most of the pieces of work reviewed in this article limit their scope to ad-hoc subsets of the SHACL specification. In addition to the custom requirements of each application, this is commonly done in order to avoid excessively complex language components. At the same time, it is often difficult to understand what these subsets exactly are as they are not always explicitly defined. Therefore, there might be scope to define reusable fragments of SHACL, that could fill the role of lightweight but expressive alternatives to the full language, similarly to how OWL fragments are defined. It might also be beneficial, for similar reasons, to converge towards a single standard or "preferred" semantics for SHACL recursion, which could be defined in a future version of the specification.

References

1. A. Amarilli and M. Benedikt and P. Bourhis and M. Vanden Boom: Query Answering with Transitive and Linear-Ordered Data. In: IJCAI'16. pp. 893–899 (2016)
2. Al Bassit, A., Krasnashchok, K., Skhiri, S., Mustapha, M.: Automated Compliance Checking with SHACL (2020)
3. Andresel, M., Corman, J., Ortiz, M., Reutter, J.L., Savkovic, O., Simkus, M.: Stable Model Semantics for Recursive SHACL. In: Proceedings of The Web Conference 2020. p. 1570–1580. WWW '20 (2020)
4. Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., Tsamoura, E.: Benchmarking the chase. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. pp. 37–52. ACM (2017)
5. Berger, R.: The Undecidability of the Domino Problem. MAMS **66**, 1–72 (1966)
6. Boneva, I., Dusart, J., Fernández Alvarez, D., Gayo, J.E.L.: Shape Designer for ShEx and SHACL Constraints. ISWC 2019 - 18th International Semantic Web Conference, Poster (Oct 2019)
7. Brickley, D., Guha, R.: RDF schema 1.1. W3C recommendation, W3C (Feb 2014), https://www.w3.org/TR/2014/REC-rdf-schema-20140225/
8. Čerāns, K., Ovčiņņikova, J., Bojārs, U., Grasmanis, M., Lāce, L., Romāne, A.: Schema-Backed Visual Queries over Europeana and other Linked Data Resources. In: ESWC2021 Poster and Demo Track (2021)
9. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering **1**(1), 146–166 (1989)

10. Cimmino, A., Fernández-Izquierdo, A., García-Castro, R.: Astrea: Automatic Generation of SHACL Shapes from Ontologies. In: The Semantic Web. pp. 497–513. Springer International Publishing, Cham (2020)
11. Corman, J., Florenzano, F., Reutter, J.L., Savkovic, O.: SHACL2SPARQL: Validating a SPARQL Endpoint against Recursive SHACL Constraints. In: International Semantic Web Conference ISWC Satellite Events. pp. 165–168 (2019)
12. Corman, J., Florenzano, F., Reutter, J.L., Savković, O.: Validating SHACL Constraints over a Sparql Endpoint. In: The Semantic Web – ISWC 2019. pp. 145–163 (2019)
13. Corman, J., Reutter, J.L., Savković, O.: Semantics and Validation of Recursive SHACL. In: The Semantic Web – ISWC 2018. pp. 318–336 (2018)
14. Cyganiak, R., Wood, D., Markus Lanthaler, G.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, W3C (2014), `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`
15. Danielski, D., Kieronski, E.: Finite Satisfiability of Unary Negation Fragment with Transitivity. In: MFCS'19. pp. 17:1–15. LIPIcs 138, Leibniz-Zentrum fuer Informatik (2019)
16. Di Ciccio, C., Ekaputra, F.J., Cecconi, A., Ekelhart, A., Kiesling, E.: Finding Non-compliances with Declarative Process Constraints Through Semantic Technologies. In: Cappiello, C., Ruiz, M. (eds.) Information Systems Engineering in Responsible Information Systems. pp. 60–74 (2019)
17. Ekaputra, F.J., Lin, X.: SHACL4P: SHACL constraints validation within Protégé ontology editor. In: 2016 International Conference on Data and Software Engineering (ICoDSE). pp. 1–6 (2016)
18. Figuera, M., Rohde, P.D., Vidal, M.E.: Trav-SHACL: Efficiently Validating Networks of SHACL Constraints. In: Proceedings of the Web Conference 2021. p. 3337–3348. WWW '21, Association for Computing Machinery, New York, NY, USA (2021)
19. Gangemi, A., Presutti, V.: Ontology Design Patterns, pp. 221–243. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
20. Jung, J., Lutz, C., Martel, M., Schneider, T.: Querying the Unary Negation Fragment with Regular Path Expressions. In: ICDT'18. pp. 15:1–18. OpenProceedings.org (2018)
21. K Soman, R.: Modelling construction scheduling constraints using shapes constraint language (SHACL). In: 2019 European Conference on Computing in Construction. pp. 351–358. University College Dublin (2019)
22. Keuchel, D., Spicher, N.: Automatic Detection of Metadata Errors in a Registry of Clinical Studies Using Shapes Constraint Language (SHACL) Graphs. Studies in health technology and informatics **281**, 372—376 (May 2021)
23. Knublauch, H., Kontokostas, D.: Shapes Constraint Language (SHACL). W3C Recommendation, W3C (2017), `https://www.w3.org/TR/shacl/`
24. Kontokostas, D., Westphal, P., Auer, S., Hellmann, S., Lehmann, J., Cornelissen, R., Zaveri, A.: Test-driven Evaluation of Linked Data Quality. In: Proceedings of the 23rd International Conference on World Wide Web. pp. 747–758. WWW '14, ACM (2014)
25. Larhrib, M., Escribano, M., Cerrada, C., Escribano, J.J.: Converting OCL and CGMES Rules to SHACL in Smart Grids. IEEE Access **8**, 177255–177266 (2020)
26. Leinberger, M., Seifer, P., Rienstra, T., Lämmel, R., Staab, S.: Deciding SHACL Shape Containment through Description Logics Reasoning. In: The Semantic Web – ISWC 2020. Springer International Publishing (2020), (*this volume*)

27. Leinberger, M., Seifer, P., Schon, C., Lämmel, R., Staab, S.: Type Checking Program Code Using SHACL. In: The Semantic Web – ISWC 2019. pp. 399–417. Springer International Publishing, Cham (2019)
28. Lieber, S., De Meester, B., Heyvaert, P., Brückmann, F., Wambacq, R., Mannens, E., Verborgh, R., Dimou, A.: Visual Notations for Viewing and Editing RDF Constraints with UnSHACLed. Semantic Web (2021), under review
29. Lieber, S., Dimou, A., Verborgh, R.: Statistics about Data Shape Use in RDF Data. In: ISWC (Demos/Industry) (2020)
30. Marnette, B.: Generalized schema-mappings: from termination to tractability. In: Proc. of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symp. on Principles of database systems. pp. 13–22. ACM (2009)
31. Musen, M.A., Protégé Team: The Protégé Project: A Look Back and a Look Forward. AI matters **1**(4), 4—12 (June 2015)
32. Nenadić, K.R., Gavrić, M.M., Đurđević, V.I.: Validation of CIM datasets using SHACL. In: 2017 25th Telecommunication Forum (TELFOR). pp. 1–4 (2017)
33. Omran, P.G., Taylor, K., Mendez, S.R., Haller, A.: Learning SHACL Shapes from Knowledge Graphs. Semantic Web (2021), *under review*
34. Pandit, H.J., O'Sullivan, D., Lewis, D.: Using Ontology Design Patterns To Define SHACL Shapes. In: 9th Workshop on Ontology Design and Patterns (WOP2018), International Semantic Web Conference (ISWC). pp. 67–71 (2018)
35. Pandit, H.J., O'Sullivan, D., Lewis, D.: Test-Driven Approach Towards GDPR Compliance. In: Semantic Systems. The Power of AI and Knowledge Graphs. pp. 19–33. Springer International Publishing, Cham (2019)
36. Pareti, P., Konstantinidis, G., Mogavero, F.: Satisfiability and Containment of Recursive SHACL (2021), arXiv preprint 2108.13063
37. Pareti, P., Konstantinidis, G., Mogavero, F., Norman, T.J.: SHACL Satisfiability and Containment. In: The Semantic Web – ISWC 2020. pp. 474–493. Springer International Publishing, Cham (2020)
38. Pareti, P., Konstantinidis, G., Norman, T.J., Şensoy, M.: SHACL Constraints with Inference Rules. In: The Semantic Web – ISWC 2019. Springer International Publishing (2019)
39. Robinson, R.: Undecidability and Nonperiodicity for Tilings of the Plane. IM **12**, 177–209 (1971)
40. Savković, O., Kharlamov, E., Lamparter, S.: Validation of SHACL Constraints over KGs with OWL 2 QL Ontologies via Rewriting. In: The Semantic Web. pp. 314–329. Springer International Publishing, Cham (2019)
41. Schaffenrath, R., Proksch, D., Kopp, M., Albasini, I., Panasiuk, O., Fensel, A.: Benchmark for Performance Evaluation of SHACL Implementations in Graph Databases. In: Rules and Reasoning. pp. 82–96. Springer International Publishing, Cham (2020)
42. Seaborne, A., Harris, S.: SPARQL 1.1 query language. W3C recommendation, W3C (Mar 2013), https://www.w3.org/TR/2013/REC-sparql11-query-20130321/
43. Šenkýř, D.: SHACL Shapes Generation from Textual Documents. In: Enterprise and Organizational Modeling and Simulation. pp. 121–130. Springer International Publishing, Cham (2019)
44. Spahiu, B., Maurino, A., Palmonari, M.: Towards Improving the Quality of Knowledge Graphs with Data-driven Ontology Patterns and SHACL. In: International Semantic Web Conference ISWC Sattelite Events. pp. 103–117 (2018)
45. Spahiu, B., Porrini, R., Palmonari, M., Rula, A., Maurino, A.: ABSTAT: Ontology-Driven Linked Data Summaries with Pattern Minimalization. In: The Semantic Web. pp. 381–395. Springer International Publishing, Cham (2016)

46. Stani, E.: Design reusable SHACL shapes and implement a linked data validation pipeline. Code4Lib Journal **45** (2019)
47. Stolk, S., McGlinn, K.: Validation of IfcOWL datasets using SHACL. In: Proceedings of the 8th Linked Data in Architecture and Construction Workshop. pp. 91–104 (2020)
48. Thornton, K., Solbrig, H., Stupp, G.S., Labra Gayo, J.E., Mietchen, D., Prud'hommeaux, E., Waagmeester, A.: Using Shape Expressions (ShEx) to Share RDF Data Models and to Guide Curation with Rigorous Validation. In: The Semantic Web. pp. 606–620. Springer International Publishing, Cham (2019)
49. W3C OWL Working Group: OWL 2 Web Ontology Language Document Overview (Second Edition). W3C recommendation, W3C (Dec 2012), https://www.w3.org/TR/2012/REC-owl2-overview-20121211/
50. Wang, H.: Proving Theorems by Pattern Recognition II. BSTJ **40**, 1–41 (1961)
51. Wright, J., Rodríguez Méndez, S.J., Haller, A., Taylor, K., Omran, P.G.: Schímatos: A SHACL-Based Web-Form Generator for Knowledge Graph Editing. In: The Semantic Web – ISWC 2020. pp. 65–80 (2020)