

Validating SHACL Constraints with Notation3 Rules

Alexander Posselt^{1,*}, Dörthe Arndt^{1,2,†}

¹Technische Universität Dresden, Germany

²ScaDS.AI – Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig, Germany

Abstract

The Shapes Constraint Language (SHACL) is a W3C recommendation to express syntactical constraints on RDF graphs. Introduced in 2017, the standard is currently being refined and extended by a W3C working group. Among the planned additions, there are recursion and rules which allow for user-defined inferencing. Adding these features to implementations relying on SPARQL is not always straight-forward while rule-based formalisms natively support both. In this paper we show that Notation3 Logic (N3), a rule-based web logic, is expressive enough to cover the core constraints of SHACL 1.0. We implement a translation of SHACL shapes to N3. We evaluate our translation on the SHACL test suite. Our implementation reaches partial conformance in 97% of the test cases and full conformance for 92%. We discuss the cases where conformance could not be reached in detail. As N3 rules are very similar to the envisioned SHACL rules, this makes N3 reasoners a particularly good choice to implement recursive SHACL constraint checking with support for SHACL rules.

Keywords

SHACL, N3, SHACL rules, RDF validation

1. Introduction

The Shapes Constraint Language (SHACL) [1] is a W3C recommendation that makes it possible to define syntactical constraints on RDF graphs. Currently, there is a W3C working group active with the goal to refine and extend the standard [2] by, among other things, recursion and the so-called SHACL rules [3], a rule format that consumes and produces RDF triples. Graph patterns containing universal variables are used to write the premise and the conclusion of an inference rule. The rules allow blank nodes in their heads, which makes them very similar to existential rules, with the exception that the latter allow predicates of arbitrary arity while SHACL rules only allow triples. SHACL rules, furthermore, come with negation as failure and with a set of built-in functions. All these features make SHACL rules very similar to Notation3 Logic (N3) [4, 5] and it is very likely, that we can directly translate from the former into the latter¹. SHACL rules are meant to be executed prior or in combination with the evaluation of the core constraints. If N3 is expressive enough to validate these, it can serve as a base for SHACL implementations supporting rules. We thus aim to answer this question in this paper.

Traditionally, constraint validation was done with the support of SPARQL [6] and this is also one of the common approaches implementing SHACL validators [7]: Constraints are rewritten into SPARQL queries which are then applied on the datagraph to detect violations. Drawbacks of this approach are the missing support for inferencing and recursion which are planned extensions of the standard [8]. With the goal of defining the semantics SHACL, several works [9, 10, 11] suggested to use rule-based reasoning for SHACL validation. As these works mainly focus on recursion, they omit details like operations of literals and IRIs. For the practical question we tackle here, these operations are crucial as only logics supporting, for example, regular expressions for strings can be used in practice. N3 comes with several built-in predicates [12] and previous work [13, 14] indicates that these are well-suited to support constraint validation in general. We aim to make this finding more concrete by directly testing

Posters, Demos, Blue Sky, and Tutorials at SEMANTiCS 2026, Sep 2026, Ghent, Belgium

*Corresponding author.

† These authors contributed equally.

✉ alexander.posselt@mailbox.tu-dresden.de (A. Posselt); doerthe.arndt@tu-dresden.de (D. Arndt)

ORCID 0000-0002-7401-8487 (D. Arndt)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

¹Note that at the time of writing, SHACL rules were still under development.

SHACL core constraints. We define a translation from SHACL constraints to N3. As N3 allows for rule-producing rules [15], the translation itself is performed in N3. The resulting rules can directly be applied on data graphs to check for constraint violations. We evaluate our implementation by testing it on the SHACL test suite [16].

The remainder of this paper is structured as follows: in Section 2 and 3 we introduce SHACL and N3, respectively. Afterwards, in Section 4, we provide the details of our translation. In Section 5 we introduce our demonstrator and discuss the evaluation we performed on the SHACL test suite. Section 6 concludes the paper.

2. SHACL

Before explaining the details of our translation, we give a short introduction to SHACL. For more details about SHACL and RDF, we refer to the specifications [1, 17].

SHACL relies on RDF. An *RDF triple* is an instance $\langle s, p, o \rangle \in (I \cup B) \times I \times (I \cup B \cup L)$ where I is the set of IRIs, B the set of blank nodes and L the set of literals. We represent blank nodes by symbols starting with $_$. We call s the *subject*, p the *predicate*, and o the *object* of the triple. We call a set of RDF triples an *RDF graph*. We call the subjects and objects of triples occurring in a graph its *nodes*; the predicates are called *edges*. RDF graphs may contain class declarations of the form $\langle s, \text{rdf:type}, o \rangle$, here, we say that s is an instance of the class o . In RDF, it is furthermore possible to express lists using the IRIs *rdf:first*, *rdf:rest*, and *rdf:nil*. In this paper we will mostly use the RDF-turtle [18] representation $(a_1 \dots a_n)$ for lists. Note that lists can also be nested.

With SHACL, we can express constraints on RDF graphs, which can be evaluated. We call the graph we want to evaluate the *data graph*. The syntax used for shapes is again RDF. A shape can be represented by an IRI or blank node. By S , we denote the set of shapes. We use it as the subject or object of a *shapes triple*. A set of shape triples form a *shapes graph*. We briefly discuss the different kinds of shape triples.

Shapes can target nodes of the data graph. We call the targeted nodes *focus nodes*. The targeting either happens because a shape is called by another shape or directly through a *target triple*. By $T := \{\text{sh:targetNode}, \text{sh:targetClass}, \text{sh:targetSubjectOf}, \text{sh:targetObjectOf}\}$ we denote the set of *target predicates*. If a shape graph contains the triple $\langle s, \text{sh:targetNode}, o \rangle$ then s targets the node o , if it contains $\langle s, \text{sh:targetClass}, o \rangle$, then s targets all nodes from the datagraph that are instances of the class o , if $\langle s, \text{sh:targetSubjectOf}, o \rangle$ or $\langle s, \text{sh:targetObjectOf}, o \rangle$, then s targets the subject or object of edge o in the datagraph, respectively.

We can have two kinds of shapes: *node shapes* and *property shapes*. With node shapes, we typically make constraints directly on focus nodes. With property shapes, we start from the focus node and then follow a predicate path in the data graph to reach another node, which we then further constrain. The predicate or predicate path we need to follow is always the object of the predicate *sh:path*. Formally, we can decide whether a shape s is a node shape or a property shape by the presence or absence of a triple $\langle s, \text{sh:path}, o \rangle$. If we encounter such a triple, s is a property shape. The object o is a *path expression*. Path expressions are taken from SPARQL [19]. We have *predicate paths*, for these o is a simple IRI, *sequence paths* expressed as lists, or one of the following: *alternative paths*, *inverse paths*, *zero-or-more paths*, *one-or-more paths* and *zero-or-one paths*. For the last group o is a blank node or IRI and for the alternative path, there is an additional triple of the form $\langle o, \text{sh:alternativePath}, l \rangle$ where l is a list of path expressions, for the others there is a triple $\langle o, p, q \rangle$ where $p \in \{\text{sh:inversePath}, \text{sh:zeroOrMorePath}, \text{sh:oneOrMorePath}, \text{sh:zeroOrOnePath}\}$ and q a property path. With node shapes and property shapes, we then define constraints on focus nodes and values, respectively. These nodes we call *value nodes*.

For the value nodes we can define different constraints using constraint components. These can for example put a restriction on the class, node kind (blank node, literal, IRI) or data type a value node belongs to (*Value Constraints*), on the number of nodes satisfying a shape (*Cardinality Constraints*), or on a range of a value (*Value Range Constraints*). Our implementation supports all *Core Constraint Components* [1, Section 4].

3. Notation3

We now have a closer look at Notation3 Logic [4, 5]. N3's syntax is based on RDF, and all RDF triples are also valid in N3. Just as in RDF, blank nodes are interpreted as existentially quantified variables. N3 allows all kinds of terms in all positions. N3, furthermore, adds universal variables and graph terms to RDF's syntax and treats lists as first-class citizens, i.e. as proper terms and not as syntactic sugar. A universal variable starts with a question mark and is, as the name suggests, universally quantified. List terms are of the form $(a_1 \dots a_n)$ where all a_i are N3 terms. We call the list $() = nil$ the *empty list*. An N3 graph is a set of N3 triples.

N3 furthermore allows the use of *rules*. A rule consists of two graph terms connected by an implication arrow $\rightarrow$. The arrow is syntactic sugar for the N3 predicate *log:implies*, which indicates a logical implication. Rules can even occur nested in a rule head, which allows us to use N3 reasoning to produce new rules. To apply a rule, we unify the universal variables in the premise of the rule with corresponding nodes, evaluate the triple containing the built-in, and retrieve triples in the rule body.

N3 comes with several built-in predicates [12] which, for example, allow string operations or comparison between datatypes. These predicates are very important to enable rule systems to fully cover SHACL's core constraints. We explain the most important predicates for our implementation.

The predicate *log:notIncludes* expresses negation as failure. In the body of a rules it is used with a blank node in subject position and the negated graph pattern as an graph term in its object. The predicate *log:collectAllIn_{N3}* makes it possible to perform aggregation within rules. The subject argument is a list consisting of three parts: (1) the aggregation variable in whose instantiations we are interested, (2) a graph pattern to be evaluated against the data graph, and (3) a variable in which we collect the instantiations of the aggregation variable found. We often use the predicate in combination with the predicates *list:length* (length of a list), *math:lessThan* (comparison between numerical values), and *log:equalTo* (comparison of two values) for counting N3 furthermore supports predicates to retrieve data (*log:dtlit*) and node type (*log:rawType*) of a value.

4. From SHACL to N3

We use N3 rules to translate from SHACL shapes to N3 graphs. Below, we provide details of this translation which we collect here as a mapping m . To avoid any issues with local scoping or renaming of blank nodes, we expect all shape graphs to be ground. In practice, that can easily be achieved by replacing all blank nodes by "fresh" IRIs.

Let $\mathcal{S}$ be a shapes graph; we define the N3 graph $m(\mathcal{S}) = \mathcal{R}$ as the smallest N3 graph that fulfils the set of conditions C . For brevity, we only show parts of C below and provide the full list of conditions in our git repository [20].

We start with conditions $P_1 - P_5$ which add path expression declarations to the N3 graph which can then be used by the rules for property paths:

- (P1) If $\langle s, sh:path, o \rangle \in \mathcal{S}$ then $\langle o, rdf:type, :PathEx \rangle \in \mathcal{R}$,
- (P2) If $\langle o, rdf:type, :PathEx \rangle \in \mathcal{R}$ and $o = (o_1, \dots, o_n)$ is a list with first element o_1 and rest list $o_r = (o_2, \dots, o_n)$, then $\langle o_1, rdf:type, :PathEx \rangle, \langle o_r, rdf:type, :PathEx \rangle \in \mathcal{R}$,
- (P3) If $\langle (o), rdf:type, :PathEx \rangle \in \mathcal{R}$, then $\langle o, rdf:type, :PathEx \rangle \in \mathcal{R}$,
- (P4) If $\langle s, sh:alternativePath, o \rangle \in \mathcal{S}$, where o is a list and p is one of its elements, then $\langle p, rdf:type, :PathEx \rangle \in \mathcal{R}$,
- (P5) If $\langle s, p, o \rangle \in \mathcal{S}$
with $p \in \{sh:inversePath, sh:zeroOrMorePath, sh:oneOrMorePath, sh:zeroOrOnePath\}$,
then $\langle o, rdf:type, :PathEx \rangle \in \mathcal{R}$.

As N3 allows the use of lists in predicate position, we can directly use each path expression as a predicate. For these, we construct rules, that is done with the productions below.

- (P6) If $\langle o, \text{rdf:type}, \text{:PathEx} \rangle \in \mathcal{R}$ and $o = (o_1, \dots, o_n)$ is a list with $n \geq 2$, where o_1 is the first element and $o_r = (o_2, \dots, o_n)$ is the rest list, then $(\langle ?x, o_1, ?y \rangle, \langle ?y, o_2, ?z \rangle) \rightarrow \langle ?x, o, ?z \rangle \in \mathcal{R}$.
- (P7) If $\langle (o), \text{rdf:type}, \text{:PathEx} \rangle \in \mathcal{R}$, then $\langle ?x, o, ?y \rangle \rightarrow \langle ?x, (o), ?y \rangle \in \mathcal{R}$.
- (P8) If $\langle s, \text{sh:alternativePath}, o \rangle \in \mathcal{S}$, where o is a list and p is one of its elements, then $\langle ?x, p, ?y \rangle \rightarrow \langle ?x, o, ?y \rangle \in \mathcal{R}$.
- (P9) If $\langle s, \text{sh:inversePath}, o \rangle \in \mathcal{S}$, then $\langle ?x, o, ?y \rangle \rightarrow \langle ?y, s, ?x \rangle \in \mathcal{R}$.
- (P10) If $\langle s, \text{sh:zeroOrMorePath}, o \rangle \in \mathcal{S}$, then
 $\langle ?x, ?p, ?y \rangle \rightarrow \langle ?x, s, ?x \rangle \in \mathcal{R}$, $\langle ?x, ?p, ?y \rangle \rightarrow \langle ?y, s, ?y \rangle \in \mathcal{R}$,
 $\langle ?x, o, ?y \rangle \rightarrow \langle ?x, s, ?y \rangle \in \mathcal{R}$, $(\langle ?x, o, ?y \rangle, \langle ?y, s, ?z \rangle) \rightarrow \langle ?x, s, ?z \rangle \in \mathcal{R}$.
- (P11) If $\langle s, \text{sh:oneOrMorePath}, o \rangle \in \mathcal{S}$, then
 $\langle ?x, o, ?y \rangle \rightarrow \langle ?x, s, ?y \rangle \in \mathcal{R}$, $(\langle ?x, o, ?y \rangle, \langle ?y, s, ?z \rangle) \rightarrow \langle ?x, s, ?z \rangle \in \mathcal{R}$.
- (P12) If $\langle s, \text{sh:zeroOrOnePath}, o \rangle \in \mathcal{S}$, then $\langle ?x, ?p, ?y \rangle \rightarrow \langle ?x, s, ?x \rangle \in \mathcal{R}$,
 $\langle ?x, ?p, ?y \rangle \rightarrow \langle ?y, s, ?y \rangle \in \mathcal{R}$, $\langle ?x, o, ?y \rangle \rightarrow \langle ?x, s, ?y \rangle \in \mathcal{R}$.

The rules construct triples with property expressions in predicate position. In the case of one-or-more path (P11), for example, the first rule detects all direct matches, the second rule recursively builds up paths of these.

To mark focus nodes, we use triples of the form $\langle s, \text{:targets}, x \rangle$, where x is a focus node of shape s .

- (T1) If $\langle s, \text{sh:targetnode}, o \rangle \in \mathcal{S}$ then $\langle s, \text{:targets}, o \rangle \in \mathcal{R}$,
- (T2) If $\langle s, \text{sh:targetClass}, o \rangle \in \mathcal{S}$ then $\langle ?x, \text{rdf:type}, o \rangle \rightarrow \langle s, \text{:targets}, ?x \rangle \in \mathcal{R}$,
- (T3) If $\langle s, \text{sh:targetSubjectOf}, o \rangle \in \mathcal{S}$ then $\langle ?x, o, ?y \rangle \rightarrow \langle s, \text{sh:targets}, ?x \rangle \in \mathcal{R}$,
- (T4) If $\langle s, \text{sh:targetObjectOf}, o \rangle \in \mathcal{S}$ then $\langle ?y, o, ?x \rangle \rightarrow \langle s, \text{:targets}, ?x \rangle \in \mathcal{R}$.

With the triple $\langle s, \text{valueNode}, v \rangle$, we represent that v is a value node of shape s .

- (V1) If $\langle s, \text{sh:path}, p \rangle \in \mathcal{S}$, then $(\langle s, \text{:targets}, ?x \rangle, \langle ?x, p, ?y \rangle) \rightarrow \langle s, \text{:valueNode}, ?y \rangle \in \mathcal{R}$,
- (V2) If $\langle s, \text{sh:path}, p \rangle \notin \mathcal{S}$, then $\langle s, \text{:targets}, ?x \rangle \rightarrow \langle s, \text{:valueNode}, ?x \rangle \in \mathcal{R}$.

Some SHACL concepts cause shapes to pass on their value node to other shapes. We add the rules:

- (T5) If $\langle s, p, o \rangle \in \mathcal{R}$ with $p \in \{\text{sh:node}, \text{sh:property}, \text{sh:not}, \text{sh:qualifiedValueShape}\}$ then $\langle s, \text{:valueNode}, ?x \rangle \rightarrow \langle o, \text{:targets}, ?x \rangle \in \mathcal{R}$,
- (T6) If $\langle s, p, o \rangle \in \mathcal{R}$ with $p \in \{\text{sh:and}, \text{sh:or}, \text{sh:xor}\}$ and o is a list and q one of its elements, then $\langle s, \text{:valueNode}, ?x \rangle \rightarrow \langle q, \text{:targets}, ?x \rangle \in \mathcal{R}$,

Having targeting and property paths in place, we define rules for each core constraint. We aim to detect *violations*. We thus define rules that are triggered if a constraint is *not* fulfilled. In that case, the rule produces a triple $\langle n, \text{:violates}, s \rangle$, where s is a shape and n is the focus node causing the violation. We exemplify this showing parts of the rules for the value constraint components:

- (VT1) If $\langle s, \text{sh:class}, c \rangle, \langle s, \text{sh:path}, p \rangle \in \mathcal{S}$ then
 $(\langle s, \text{:targets}, ?y \rangle, \langle ?y, p, ?x \rangle, \langle _, \text{log:notIncludes}, \langle ?x, \text{rdf:type}, c \rangle \rangle) \rightarrow \langle ?y, \text{:violates}, s \rangle \in \mathcal{R}$,
- (VT2) If $\langle s, \text{sh:class}, c \rangle \in \mathcal{S}$ and $\langle s, \text{sh:path}, p \rangle \notin \mathcal{S}$ then
 $(\langle s, \text{:targets}, ?x \rangle, \langle _, \text{log:notIncludes}, \langle ?x, \text{rdf:type}, c \rangle \rangle) \rightarrow \langle ?x, \text{:violates}, s \rangle \in \mathcal{R}$,
- (VT3) If $\langle s, \text{sh:datatype}, d \rangle, \langle s, \text{sh:path}, p \rangle \in \mathcal{S}$ then
 $(\langle s, \text{:targets}, ?y \rangle, \langle ?y, p, ?x \rangle, \langle _ \text{?} c, \text{log:dtlit}, ?x \rangle, \langle ?c, \text{log:notEqualTo}, d \rangle) \rightarrow \langle ?y, \text{:violates}, s \rangle \in \mathcal{R}$
- (VT4) If $\langle s, \text{sh:datatype}, d \rangle \in \mathcal{S}$ and $\langle s, \text{sh:path}, p \rangle \notin \mathcal{S}$ then
 $(\langle s, \text{:targets}, ?x \rangle, \langle _ \text{?} c, \text{log:dtlit}, ?x \rangle, \langle ?c, \text{log:notEqualTo}, d \rangle) \rightarrow \langle ?x, \text{:violates}, s \rangle \in \mathcal{R}$

The rules we generate depend on N3's built-in functions and negation. The rules produced by VT1 and VT2 use negation as failure to check for the absence of a class declaration ($\langle ?x, \text{rdf:type}, c \rangle$), the rules for datatype checking (VT3, VT4), use N3's built-in predicate *log:dtlit* to check for a literal's datatype.

5. Implementation & Evaluation

We implemented the above translation. All code is available at our git repository [20]. We host a demonstrator at <https://kilanopi.github.io/SHACLN3>.

We perform the translation detailed in the previous section in N3 taking advantage N3's support for rule-producing rules. For rule production VT1, we for example use the rule

$$(\langle ?s, \text{sh:class}, ?c \rangle, \langle ?s, \text{sh:path}, ?p \rangle) \rightarrow ((\langle ?s, \text{:targets}, ?y \rangle, \langle ?y, ?p, ?x \rangle, \langle _, \text{log:notIncludes}, \langle ?x, \text{rdf:type}, ?c \rangle \rangle) \rightarrow \langle ?y, \text{:violates}, ?s \rangle)$$

We use the EYE Reasoner [21, 22] to first ground the shapes graph and to afterwards apply the rules ("Translation" in the demonstrator). From the shape ($\langle :S1, \text{sh:class}, \text{:Person} \rangle, \langle :S1, \text{sh:path}, \text{:spouse} \rangle$), we for example get the rule²

$$((\langle :S1, \text{:targets}, ?y \rangle, \langle ?y, \text{:spouse}, ?x \rangle, \langle _, \text{log:notIncludes}, \langle ?x, \text{rdf:type}, \text{:Person} \rangle \rangle) \rightarrow \langle ?y, \text{:violates}, :S1 \rangle)$$

The generated rules can be exchanged and combined with other N3 rules. Executing the rules on the data graph with an N3 reasoner is expected to produce the same results as validating the graph against the shapes using a SHACL engine. In our demonstrator we again employ the EYE reasoner ("Validation").

To ensure that executing the translated rules on the datagraph provides the same result as the validation with a SHACL validator, we evaluated our implementation on the SHACL test suite [16]. This test suite was created by the SHACL working group and contains 97 tests for core constraints. SHACL validators can use the test suite to conform full or partial compliance. Partial compliance means that the validator provides the correct result – conformance of the data graph to a shapes graph is true or false – full compliance means that it additionally produces a correct validation report.

We performed our tests with EYE (v11.24.4) and SWI Prolog (9.3.6). For 89 of the 97 test cases, we could show full compliance, for 5 cases we had partial compliance and for 3 test cases our tests failed.³ We discuss the critical cases beginning with the failures.

We list the failing test cases by their names, these were *shacl-shacl*, *lessThan-002* and *uniqueLang-002*. In the first test, the datagraph is itself a SHACL shape and the shapes graph for validation tests, among other things, for the well-formedness of an RDF-list. N3 treats lists between brackets as first-class-citizens while this notation is considered syntactic sugar for first-rest pairs in RDF. The datagraph of the test contains a turtle list but has conditions on the list's first-rest representation. To cover that case, additional rules would need to be added. The failure on *lessThan-002* was caused by a mismatch between the string comparison functions of SHACL and N3 when a string is compared to a number. The test *uniqueLang-002* fails because the EYE reasoner performs RDF's D-Entailment during our grounding step and the test case contains the literal `"1"^^xsd:boolean` which in SHACL is not the same as `"true"^^xsd:boolean`.

In two cases (*uniqueLang-001*, *property-001*) we only have partial compliance because the validation report expects duplicates of triples which is against N3's (and RDF's) set view of graphs. The shapes of the test cases *path-strange-001*, and *path-strange-002* both have one single blank node representing two different path expressions of which one is a list written as first-rest pairs. This combination is problematic for N3's treatment of lists. For *minInclusive-003* we do not have full conformance because EYE's implementation of the N3 function to compare time expressions like `"2002-10-10T12:00:00-05:00"8sd:dateTime` ignores the difference part `-05:00`.

²Note that in this paper we used an abbreviated syntax, the demonstrator produces proper N3.

³We provide a full table displaying the test results in our github repository.

6. Conclusion

In this paper we defined a translation from SHACL to N3 rules. We provided an implementation and a demonstrator performing this translation. We validated the implementations using the test cases from the SHACL test suite. The goal of this work was to study the close relation between N3 and SHACL, especially if the latter is enhanced by SHACL rules, and to test whether N3 and the existing reasoning infrastructure for this language could be employed to combine validation and rule-based reasoning. As it is clear that N3, as a rule logic on RDF, can most likely support SHACL rules (depending on how the final state of the SHACL rules specification will look like), we focused on the core constraints. We found that for 97% of the test cases we at least reached partial compliance. The missing 3% were mainly caused by differences between the frameworks. To avoid the side effects of D-Entailment as we had it with the EYE reasoner, an implementation where this entailment can be disabled can be used, if a use case really relies on this. The problems resulting from differences in built-in predicates need to be addressed by the N3 community group (just as SHACL rules, N3 is under development), we hope that our work provides insights for this discussion. The problems caused by the different understandings of lists can be fixed by adding explicit rules to decompose the lists during validation. We do not see any of these cases as a major problem for practical use cases and thus conclude that SHACL constraint checking can be performed with N3.

As future work, we want to improve the validation reports to reach a better coverage full compliance (currently 92% of all tests), we plan to do performance tests and to automatically translate from SHACL rules to N3. The latter of course depends on the final state of this standard.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

- [1] H. Knublauch, D. Kontokostas, Shapes Constraint Language (SHACL), Recommendation, World Wide Web Consortium (W3C), 2017. URL: <https://www.w3.org/TR/shacl/>.
- [2] H. Knublauch, T. Bergwinkl, Y. Taghzouti, J. Wright (Eds.), SHACL 1.2 Core, W3C Working Draft, 2026. Available at <https://www.w3.org/TR/shacl12-core/>.
- [3] R. David, D. Habgood, A. Seaborne, S. Steyskal (Eds.), SHACL 1.2 Rules, W3C Working Draft, 2026. Available at <https://www.w3.org/TR/shacl12-rules/>.
- [4] W. V. Woensel, D. Arndt, P.-A. Champin, D. Tomaszuk, G. Kellogg, Notation3 language, 2026. <https://w3c.github.io/N3/reports/20230703/>.
- [5] T. Berners-Lee, D. Connolly, L. Kagal, Y. Scharf, J. Hendler, N3Logic: A logical framework for the World Wide Web, *Theory and Practice of Logic Programming* (2008) 249–269. doi:10.1017/S1471068407003213.
- [6] T. Hartmann, B. Zapilko, J. Wackerow, K. Eckert, Validating RDF data quality using constraints to direct the development of constraint languages, in: *Tenth IEEE International Conference on Semantic Computing, ICSC 2016, Laguna Hills, CA, USA, February 4-6, 2016*, IEEE Computer Society, 2016, pp. 116–123. URL: <https://doi.org/10.1109/ICSC.2016.43>.
- [7] P. Pareti, G. Konstantinidis, A review of SHACL: from data validation to schema reasoning for RDF graphs, in: M. Simkus, I. Varzinczak (Eds.), *Reasoning Web. Declarative Artificial Intelligence - 17th International Summer School 2021, Leuven, Belgium, September 8-15, 2021, Tutorial Lectures, Lecture Notes in Computer Science*, Springer, 2021, pp. 115–144. URL: https://doi.org/10.1007/978-3-030-95481-9_6. doi:10.1007/978-3-030-95481-9_6.
- [8] Data Shapes Working Group Charter, 2024. Available at <https://www.w3.org/2024/12/data-shapes.html>.

- [9] J. Corman, J. L. Reutter, O. Savkovic, Semantics and validation of recursive SHACL, in: D. Vrandečić, K. Bontcheva, M. C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L. Kaffee, E. Simperl (Eds.), *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference*, Monterey, CA, USA, October 8-12, 2018, *Proceedings, Part I, Lecture Notes in Computer Science*, Springer, 2018, pp. 318–336. URL: https://doi.org/10.1007/978-3-030-00671-6_19. doi:10.1007/978-3-030-00671-6_19.
- [10] M. Andresel, J. Corman, M. Ortiz, J. L. Reutter, O. Savkovic, M. Simkus, Stable model semantics for recursive SHACL, in: Y. Huang, I. King, T. Liu, M. van Steen (Eds.), *WWW '20: The Web Conference 2020*, Taipei, Taiwan, April 20-24, 2020, *ACM / IW3C2*, 2020, pp. 1570–1580. URL: <https://doi.org/10.1145/3366423.3380229>. doi:10.1145/3366423.3380229.
- [11] C. Okulmus, M. Simkus, SHACL validation under the well-founded semantics, in: P. Marquis, M. Ortiz, M. Pagnucco (Eds.), *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning*, KR 2024, Hanoi, Vietnam. November 2-8, 2024, 2024. URL: <https://doi.org/10.24963/kr.2024/52>. doi:10.24963/kr.2024/52.
- [12] W. V. Woensel, P. Hochstenbach, Notation3 builtin functions, 2023. <https://w3c-cg.github.io/N3/reports/20230703/builtins.html>.
- [13] De Meester, Heyvaert, Arndt, Dimou, Verborgh, Rdf graph validation using rule-based reasoning, *Semantic Web Journal* 12 (2020) 117–142. URL: <https://content.iospress.com/articles/semantic-web/sw200384>. doi:10.3233/SW-200384.
- [14] Arndt, De Meester, Dimou, Verborgh, Mannens, Using rule-based reasoning for rdf validation, in: *Rules and Reasoning: International Joint Conference, RuleML+RR 2017*, London, UK, July 12–15, 2017, volume 10364 of *Lecture Notes in Computer Science*, Springer, Cham, 2017, pp. 22–36. doi:10.1007/978-3-319-61252-2_3.
- [15] D. Arndt, B. D. Meester, P. Bonte, J. Schaballie, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. D. Turck, R. V. de Walle, E. Mannens, Improving OWL RL reasoning in N3 by using specialized rules, in: V. Tamma, M. Dragoni, R. S. Gonçalves, A. Lawrynowicz (Eds.), *Ontology Engineering - 12th International Experiences and Directions Workshop on OWL, OWLED 2015*, co-located with ISWC 2015, Bethlehem, PA, USA, October 9-10, 2015, *Revised Selected Papers, Lecture Notes in Computer Science*, Springer, 2015, pp. 93–104. URL: https://doi.org/10.1007/978-3-319-33245-1_10. doi:10.1007/978-3-319-33245-1_10.
- [16] H. Knublauch, D. Kontokostas, J. E. Labra Gayo, Shacl test suite, 2026. <https://w3c.github.io/data-shapes/data-shapes-test-suite/>.
- [17] R. Cyganiak, D. Wood, M. Lanthaler (Eds.), *RDF 1.1 Concepts and Abstract Syntax*, W3C Recommendation, 2014. Available at <http://www.w3.org/TR/rdf11-concepts/>.
- [18] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, G. Carothers, *Turtle - Terse RDF Triple Language*, w3c Recommendation, 2014. <http://www.w3.org/TR/turtle/>.
- [19] S. Harris, A. Seaborne, *SPARQL 1.1 Query Language*, W3C Recommendation, W3C, 2013. URL: <https://www.w3.org/TR/2013/REC-sparql11-query-20130321/>.
- [20] A. Posselt, D. Arndt, SHACL validation with N3 Logic, 2026. URL: <https://github.com/Kilanopi/ShaclN3>.
- [21] J. De Roo, Euler Yet another proof Engine - EYE, 2026. URL: <https://github.com/eyereasoner/eye/>.
- [22] R. Verborgh, J. De Roo, Drawing Conclusions from Linked Data on the Web: The EYE Reasoner, *IEEE Software* (2015) 23–27. doi:10.1109/MS.2015.63.