Formalization of SHACL

Maxime Jakubowski

Universiteit Hasselt, Hasselt, Belgium
maxime.jakubowski@uhasselt.be

June 2021

In this document, we present our SHACL formalization, building on the work by Corman, Reutter and Savkovic [1]. We extend their formalization to cover all features of SHACL, such as disjointness, zero-or-one property paths, closedness, language tags, node tests, and literals.

1 SHACL formalization

We assume three pairwise disjoint infinite sets I, L, and B of *IRIs*, *literals*, and *blank nodes*, respectively. We use N to denote the union $I \cup B \cup L$; all elements of N are referred to as *nodes*. Literals may have a "language tag" [2]. We abstract this by assuming an antireflexive relation $\sim$ on L, where $l \sim l'$ represents that l and l' are distinct literals with the same language tag. Moreover, we assume a strict partial order $<$ on L that abstracts comparisons between numeric values, strings, dateTime values, etc.

An *RDF triple* (s, p, o) is an element of $(I \cup B) \times I \times N$. We refer to the elements of the triple as the subject s, the property p, and the object o. An *RDF graph* G is a finite set of RDF triples.

We formalize SHACL property paths as *path expressions* E. Their syntax is given by the following grammar, where p ranges over I:

$$E ::= p \mid E^- \mid E_1/E_2 \mid E_1 \cup E_2 \mid E^* \mid E?$$

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

Table 1: Evaluation of path expressions.

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

The formal syntax of *shapes* ϕ is now given by the following grammar:

$$\begin{aligned}\phi ::= \top \mid \bot \mid \mathit{hasShape}(s) \mid \mathit{test}(t) \mid \mathit{hasValue}(c) \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \\ \mid \geq_n E.\phi \mid \leq_n E.\phi \mid \forall E.\phi \mid \mathit{eq}(E,p) \mid \mathit{disj}(E,p) \mid \mathit{closed}(P) \\ \mid \mathit{lessThan}(E,p) \mid \mathit{lessThanEq}(E,p) \mid \mathit{uniqueLang}(E)\end{aligned}$$

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

We formalize SHACL shapes graphs as *schemas*. We first define the notion of *shape definition*, as a triple (s, ϕ, τ) where $s \in I \cup B$, and ϕ and τ are shapes. The elements of the triple are referred to as the *shape name*, the *shape expression*, and the *target expression*, respectively.

Now a *schema* is a finite set H of shape definitions such that no two shape definitions have the same shape name. Moreover, we consider only *nonrecursive* schemas. Here, H is said to be recursive if there is a directed cycle in the directed graph formed by the shape names, with an edge $s_1 \to s_2$ if $\mathit{hasShape}(s_2)$ occurs in the shape expression defining s_1.

In order to define the semantics of shapes and shape schemas, we first recall that a path expression E evaluates on an RDF graph G to a binary relation on N, denoted by $[\![E]\!]^G$ and defined in Table 1.

We are now ready to define when a node a *conforms* to a shape ϕ in a graph G, in the context of a schema H, denoted by $H, G, a \models \phi$. For the boolean operators $\top$ (true), $\bot$ (false), $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), the definition is obvious. For the other constructs, the definition is shown in Table 2. In the definition for $\mathit{hasShape}(s)$ we use the notation $\mathit{def}(s, H)$ to denote the shape expression defining shape name s in H. If s does not have a definition in H, we let $\mathit{def}(s, H)$ be $\top$ (this is the behavior in real SHACL). We use the notation $R(x)$, for a binary relation R, to denote the set $\{y \mid (x,y) \in R\}$. We apply this notation to the case where R is of the form $[\![E]\!]^G$ and x is a node. We also use the notion $\sharp X$ for the cardinality of a set X. Note that the conditions for $\mathit{lessThan}(E,p)$ and $\mathit{lessThanEq}(E,p)$ imply that b and c must be literals.

Table 2: Conditions for conformance of a node to a shape.

ϕ	$H, G, a \models \phi$ if:
$hasValue(c)$	$a = c$
$test(t)$	a satisfies t
$hasShape(s)$	$H, G, a \models def(s, H)$
$\geq_n E.\psi$	$\sharp\{b \in [\![E]\!]^G(a) \mid H, G, b \models \psi\} \geq n$
$\leq_n E.\psi$	$\sharp\{b \in [\![E]\!]^G(a) \mid H, G, b \models \psi\} \leq n$
$\forall E.\psi$	every $b \in [\![E]\!]^G(a)$ satisfies $H, G, b \models \psi$
$eq(E, p)$	the sets $[\![E]\!]^G(a)$ and $[\![p]\!]^G(a)$ are equal
$disj(E, p)$	the sets $[\![E]\!]^G(a)$ and $[\![p]\!]^G(a)$ are disjoint
$closed(P)$	for all triples $(a, p, b) \in G$ we have $p \in P$
$lessThan(E, p)$	$b < c$ for all $b \in [\![E]\!]^G(a)$ and $c \in [\![p]\!]^G(a)$
$lessThanEq(E, p)$	$b \leq c$ for all $b \in [\![E]\!]^G(a)$ and $c \in [\![p]\!]^G(a)$
$uniqueLang(E)$	$b \nsim c$ for all $b, c \in [\![E]\!]^G(a)$.

Finally, we can define conformance of a graph to a schema as follows. RDF graph G *conforms* to schema H if for every shape definition $(s, \phi, \tau) \in H$ and for every $a \in N$ such that $H, G, a \models \tau$, we have $H, G, a \models \phi$.

In the next section, we show that our formalization fully covers real (nonrecursive) SHACL.

2 Translating real SHACL to formal SHACL

In this section we define the function t which maps a SHACL shapes graph $\mathcal{S}$ to a schema H.

Assumptions about the shapes graph:

- All shapes of interest must be explicitly declared to be a `sh:NodeShape` or `sh:PropertyShape`
- The shapes graph is well-formed

Let the sets $\mathcal{S}_n$ and $\mathcal{S}_p$ be the sets of all node shape shape names, respectively property shape shape names defined in the shapes graph $\mathcal{S}$. Let d_x denote the set of RDF triples in $\mathcal{S}$ with x as the subject. We define $t(\mathcal{S})$ as follows:

$$t(\mathcal{S}) = \{(x, t_{nodeshape}(d_x), t_{target}(d_x)) \mid x \in \mathcal{S}_n\} \cup \{(x, t_{propertyshape}(d_x), t_{target}(d_x)) \mid x \in \mathcal{S}_p\}$$

where we define $t_{nodeshape}(d_x)$ in Section 2.2, $t_{propertyshape}(d_x)$ in Section 2.3 and $t_{target}(d_x)$ in Section 2.4.

2.1 Defining $t_{path}(p)$

This function translates the Property Paths to path expressions. This part of the translation deals with the SHACL keywords:

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

We define $t_{path}(p)$ as follows:

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

2.2 Defining $t_{nodeshape}(d_x)$

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

$$t_{shape}(d_x) \wedge t_{logic}(d_x) \wedge t_{tests}(d_x) \wedge t_{value}(d_x) \wedge t_{in}(d_x) \wedge t_{closed}(d_x)$$

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

2.2.1 Defining $t_{shape}(d_x)$

This function translates the Shape-based Constraint Components from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:node` and `sh:property`.

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

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

2.2.2 Defining $t_{logic}(d_x)$

This function translates the Logical Constraint Components from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:and`, `sh:or`, `sh:not`, `sh:xone`.

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

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

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

2.2.3 Defining $t_{tests}(d_x)$

This function translates the Value Type Constraint Components, Value Range Constraint Components, and String-based Constraint Components, with exception to the `sh:languageIn` keyword which is handled in Section 2.3.5, from d_x to shapes from the formalization. This function covers the SHACL keywords:

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

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

$$t_{tests'}(d_x) \wedge \bigwedge_{(x,\texttt{sh:class},y)\in d_x} \geq_1 \texttt{rdf:type}.hasValue(y)$$

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

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

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

2.2.4 Defining other constraint components

These functions translate the Other Constraint Components from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:closed`, `sh:ignoredProperties`, `sh:hasValue`, `sh:in`.

We define the following functions:

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

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

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

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

2.3 Defining $t_{propertyshape}(d_x)$

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

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

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

2.3.1 Defining $t_{card}(E, d_x)$

This function translates the Cardinality Constraint Components. from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:minCount`, `sh:maxCount`.

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

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

2.3.2 Defining $t_{pair}(E, d_x)$

This function translates the Property Pair Constraint Components from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:equals`, `sh:disjoint`, `sh:lessThan`. `sh:lessThanOrEquals`.

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

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

2.3.3 Defining $t_{qual}(E, d_x)$

This function translates the (Qualified) Shape-based Constraint Components from d_x to shapes from the formalization. This function covers the SHACL keywords:

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

We distinguish between the case where the parameter `sh:qualifiedValueShapesDisjoint` is set to *true*, and the case where it is not:

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

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

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

We also define:

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

We now define

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

and

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

2.3.4 Defining $t_{all}(E, d_x)$

This function translates the constraint components that are not specific to property shapes, but which are applied on property shapes.

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

$$\forall E.(t_{shape}(d_x) \wedge t_{logic}(d_x) \wedge t_{tests}(d_x) \wedge t_{in}(d_x) \wedge t_{closed}(d_x)) \wedge \geq_1 E.t_{value}(d_x)$$

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

2.3.5 Defining $t_{lang}(E, d_x)$

This function translates the constraint components Unique Lang Constraint Component and Language In Constraint Component from d_x to shapes from the formalization. This function covers the SHACL keywords: `sh:uniqueLang` and `sh:languageIn`.

We define

$$t_{lang}(E, d_x) = t_{languagein}(E, d_x) \wedge t_{uniquelang}(E, d_x)$$

where $t_{languagein}(E, d_x)$ and $t_{uniquelang}(E, d_x)$ are defined next.

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

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

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

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

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

2.4 Defining $t_{target}(d_x)$

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

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

We define the function as follows:

$$t_{target}(d_x) = \begin{cases} hasValue(y) & \text{if } (x, \texttt{sh:targetNode}, y) \in d_x \\ \geq_1 \texttt{rdf:type}.hasValue(y) & \text{if } (x, \texttt{sh:targetClass}, y) \in d_x \\ \geq_1 y.\top & \text{if } (x, \texttt{sh:targetSubjectsOf}, y) \in d_x \\ \geq_1 y^-.\top & \text{if } (x, \texttt{sh:targetObjectsOf}, y) \in d_x \\ \bot & \text{otherwise} \end{cases}$$

References

[1] J. Corman, J.L. Reutter, and O. Savkovic. Semantics and validation of recursive SHACL. In D. Vrandecic et al., editors, *Proceedings 17th International Semantic Web Conference*, volume 11136 of *Lecture Notes in Computer Science*, pages 318–336. Springer, 2018. Extended version, technical report KRDB18-01, `https://www.inf.unibz.it/krdb/tech-reports/`.

[2] RDF 1.1 primer. W3C Working Group Note, June 2014.