Formalization of SHACL

Maxime Jakubowski

Hasselt University

June 10, 2021

1 The shapes language

We assume three pairwise disjoint infinite sets I, L and B of IRIs, literals and blank nodes respectively. We use N to denote the set of all nodes $I \cup B \cup L$. We define the relation $\sim$ to be an equivalence relation over the set of literals L such that for two literals $l, l' \in L$ we have $l \sim l'$ if and only if they both have a language tag and those language tags are the same.

An *RDF triple* (s, p, o) is an element of $(I \cup B) \times I \times (I \cup B \cup L)$. We refer to the elements of the triple as the subject s, predicate p and object o.

An *RDF graph* G is a finite set of RDF triples. The set of all subjects and objects occurring in the graph is referred to as the nodes of the graph.

A *path expression* E is given by the following grammar:

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

with $p \in I$ representing a property name.

We assume an infinite set Ω of tests on nodes in the graph, e.g., node type tests or pattern matching test. A *shape* ϕ is given by the following grammar:

$$\begin{aligned}\phi ::= {} & \top \mid \bot \mid \mathit{hasShape}(s) \mid \mathit{test}(t) \mid \mathit{hasValue}(c) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \\ & \geq_n E.\phi_1 \mid\leq_n E.\phi_1 \mid \forall E.\phi \mid \mathit{eq}(E,p) \mid \mathit{disj}(E,p) \mid \mathit{closed}(P) \mid \\ & \mathit{lessThan}(E,p) \mid \mathit{lessThanEq}(E,p) \mid \mathit{uniqueLang}(E)\end{aligned}$$

with $s \in I \cup B$, $c, p \in I$, $t \in \Omega$ and $P \subseteq I$ representing a set of property names. We call the language described by this grammar $\mathcal{L}$.

A *shape definition* is a triple (s, ϕ, τ) with $s \in I \cup B$, and ϕ, τ shapes. The elements of the triple are referred to as the *shape name*, *shape expression*, and the *target expression* respectively.

A *schema* is a finite set of shape definitions. For the purpose of this work, we only consider schemas that are non-recursive. Note: add a definition of a non-recursive schema.

The *evaluation* of a shape, together with a shape schema, in an RDF graph results in a subset of N that satisfies the constraints expressed in the shape.

E	$[\![E]\!]^G$
$[\![E]\!]^G$	$\{(s,o) \mid \exists p : (s,p,o) \in G\}$
$[\![E^-]\!]^G$	$\{(o,s) \in N^2 \mid (s,o) \in [\![E]\!]^G\}$
$[\![E?]\!]^G$	$\{(a,a) \mid a \in N\} \cup [\![E]\!]^G$
$[\![E_1 \cdot E_2]\!]^G$	$\{(s,o) \in N^2 \mid (s,q) \in [\![E_1]\!]^G \wedge (q,o) \in [\![E_2]\!]^G\}$
$[\![E^*]\!]^G$	the reflexive, transitive closure of $[\![E]\!]^G$
$[\![E_1 \cup E_2]\!]^G$	$[\![E_1]\!]^G \cup [\![E_2]\!]^G$

Table 1: Evaluation of a path expression

First, the evaluation of a path expression E in an RDF graph G is defined in Table 1.

We define the conformance relation $\models$ specifying when a node $a \in N$, given a graph G and a shape schema H, *conforms* to a shape ϕ:

- $H, G, a \models \top$ holds for all $a \in N$;
- $H, G, a \models \mathit{hasValue}(a)$ holds for all $a \in N$;
- $H, G, a \models \mathit{test}(t)$ iff a satisfies test t;
- $H, G, a \models \mathit{hasShape}(s)$ iff $H, G, a \models \phi_1$ with $(s, \phi_1, \tau) \in H$;
- $H, G, a \models \neg\phi_1$ iff $H, G, a \not\models \phi_1$;
- $H, G, a \models \phi_1 \wedge \phi_2$ iff $H, G, a \models \phi_1$ and $H, G, a \models \phi_2$;
- $H, G, a \models \phi_1 \vee \phi_2$ iff $H, G, a \models \phi_1$ or $H, G, a \models \phi_2$;
- $H, G, a \models\ \geq_n E.\phi_1$ iff there exist at least n nodes $b_1, \ldots, b_n$ such that $(a, b_i) \in [\![E]\!]^G$ and $H, G, b_i \models \phi_1$ with $1 \leq i \leq n$;
- $H, G, a \models\ \leq_n E.\phi_1$ iff there exist at most n nodes $b_1, \ldots, b_n$ such that $(a, b_i) \in [\![E]\!]^G$ and $H, G, b_i \models \phi_1$ with $1 \leq i \leq n$;
- $H, G, a \models \forall E.\phi_1$ iff for all b when $(a, b) \in [\![E]\!]^G$, then $H, G, b \models \phi_1$;
- $H, G, a \models \mathit{eq}(E, p)$ iff $[\![p]\!]^G(a)$ and $[\![E]\!]^G(a)$ are equal;
- $H, G, a \models \mathit{disj}(E, p)$ iff $[\![p]\!]^G(a)$ and $[\![E]\!]^G(a)$ are disjoint;
- $H, G, a \models \mathit{closed}(P)$ iff for all triples $(a, p, b) \in G$ we have $p \in P$;
- $H, G, a \models \mathit{lessThan}(E, p)$ iff the maximal element from $[\![E]\!]^G(a)$ is strictly smaller than the minimal element from $[\![E]\!]^G(a)$;
- $H, G, a \models \mathit{lessThanEq}(E, p)$ iff the maximal element from $[\![E]\!]^G(a)$ is smaller or equal to the minimal element from $[\![E]\!]^G(a)$;
- $H, G, a \models \mathit{uniqueLang}(E)$ iff for every two nodes b, c from $[\![E]\!]^G(a)$ we have $(b, c) \notin\ \sim$.

An RDF graph G validates against a schema H when for every shape definition $(s, \phi, \tau) \in H$ we have that for all $a \in N$ if $H, G, a \models \tau$ then $H, G, a \models \phi$.

2 From SHACL to the shapes language

In this section we define the function t which maps a SHACL shapes graph L to a shape 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 L_n and L_p be the sets of all NodeShape shape names, and PropertyShape shape names defined in the shapes graph L. Let d_x denote the set of RDF triples with x as the subject. We define t as follows:

$$t(L) = \bigcup_{x \in L_n} \{(x, t_{nodeshape}(d_x), t_{target}(d_x))\} \cup \bigcup_{x \in L_p} \{(x, t_{propertyshape}(d_x), t_{target}(d_x))\}$$

3 Defining $t_{path}(p)$

This function translates the Property Paths. Keywords: `sh:inversePath`, `sh:alternativePath`, `sh:zeroOrMorePath`, `sh:oneOrMorePath`, `sh:zeroOrOnePath`, `sh:alternativePath`.

Definition:

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

4 Defining $t_{nodeshape}(d_x)$

Definition:

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

4.1 Defining $t_{shape}(d_x)$

This function translates the Shape-based Constraint Components. Keywords: `sh:node`, `sh:property`.

Definition:

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

4.2 Defining $t_{logic}(d_x)$

This function translates the Logical Constraint Components. Keywords: `sh:and`, `sh:or`, `sh:not`, `sh:xone`.

Definition:

$$\begin{aligned} t_{logic}(d_x) = & \bigwedge_{(x,\texttt{sh:and},y)\in d_x} (\bigwedge \mathit{hasShape}(y)) \wedge \bigwedge_{(x,\texttt{sh:or},y)\in d_x} (\bigvee \mathit{hasShape}(y)) \wedge \\ & \bigwedge_{(x,\texttt{sh:xone},y)\in d_x} (\bigvee_{a\in y} (a \wedge \bigwedge_{b\in y-\{a\}} \neg \mathit{hasShape}(b))) \wedge \\ & \bigwedge_{(x,\texttt{sh:not},y)\in d_x} (\neg \mathit{hasShape}(y)) \end{aligned}$$

4.3 Defining $t_{tests}(d_x)$

This function translates the Value Type Constraint Components, Value Range Constraint Components, and String-based Constraint Components. Keywords: `sh:class`, `sh:datatype`, `sh:nodeKind`, `sh:minExclusive`, `sh:maxExclusive`, `sh:minInclusive`, `sh:maxInclusive`, `sh:minLength`, `sh:maxLength`, `sh:pattern`, `sh:languageIn`.

Definition:

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

We define $t_{tests'}$ to be a conjunction of shapes of the form $\mathit{test}(o)$ where o represents an element from the set Ω that corresponds to the constraints defined by the constraint component. This applies to the following keywords: : `sh:class`, `sh:datatype`, `sh:nodeKind`, `sh:minExclusive`, `sh:maxExclusive`, `sh:minInclusive`, `sh:maxInclusive`, `sh:minLength`, `sh:maxLength`, `sh:pattern`. The values of `sh:languageIn` are SHACL lists and are each translated to a disjunction of tests on language tags.

4.4 Defining other constraint components

These functions translate the Other Constraint Components. Keywords: `sh:closed`, `sh:ignoredProperties`, `sh:hasValue`, `sh:in`.

Definition:

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

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

Let $P = \{p \mid \exists y : (x, \texttt{sh:property}, y), (y, \texttt{sh:path}, p) \in d_x \wedge p \text{ is a property name}\} \cup \bigcup_{\{y|(x,\texttt{sh:ignoredProperties},y)\in d_x\}} y$

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

5 Defining $t_{propertyshape}(d_x)$

Definition: Let p be the property path associated with d_x. Let E be $t_{path}(p)$.

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

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

This function translates the Cardinality Constraint Components. Keywords: `sh:minCount`, `sh:maxCount`.

Definition:

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

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

This function translates the Property Pair Constraint Components. Keywords: `sh:equals`, `sh:disjoint`, `sh:lessThan`. `sh:lessThanOrEquals`.

Definition:

$$\begin{aligned} t_{pair}(E, d_x) = & \bigwedge_{(x,\texttt{sh:equals},p)\in d_x} \mathit{eq}(E, p) \wedge \bigwedge_{(x,\texttt{sh:disjoint},p)\in d_x} \mathit{disj}(E, p) \wedge \\ & \bigwedge_{(x,\texttt{sh:lessThan},p)\in d_x} \mathit{lessThan}(E, p) \wedge \\ & \bigwedge_{(x,\texttt{sh:lessThanOrEquals},p)\in d_x} \mathit{lessThanEq}(E, p) \end{aligned}$$

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

This function translates the (Qualified) Shape-based Constraint Components. Keywords: `sh:qualifiedValueShape`, `sh:qualifiedMinCount`, `sh:qualifiedMaxCount`. `sh:qualifiedValueShapesDisjoint`.

Definition:

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

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

$$\begin{aligned} t_{sibl}(E, d_x) = & \bigwedge_{(x,\texttt{sh:qualifiedValueShape},y)\in d_x} \bigwedge_{(x,\texttt{sh:qualifiedMinCount},z)\in d_x} \\ & (\geq_z E.(hasShape(y) \wedge \bigwedge_{s \in sibl} \neg hasShape(s)) \wedge \\ & \bigwedge_{(x,\texttt{sh:qualifiedValueShape},y)\in d_x} \bigwedge_{(x,\texttt{sh:qualifiedMaxCount},z)\in d_x} \\ & (\leq_z E.(hasShape(y) \wedge \bigwedge_{s \in sibl} \neg hasShape(s)) \end{aligned}$$

$$\begin{aligned} t_{nosibl}(E, d_x) = & \bigwedge_{(x,\texttt{sh:qualifiedValueShape},y)\in d_x} \bigwedge_{(x,\texttt{sh:qualifiedMinCount},z)\in d_x} (\geq_z E.hasShape(y)) \wedge \\ & \bigwedge_{(x,\texttt{sh:qualifiedValueShape},y)\in d_x} \bigwedge_{(x,\texttt{sh:qualifiedMaxCount},z)\in d_x} (\leq_z E.hasShape(y)) \end{aligned}$$

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

This function translates the NodeShape constraint components that are applied on PropertyShapes.

Definition:

$$\begin{aligned} t_{all}(E, d_x) = & \forall E.(t_{shape}(d_x) \wedge t_{logic}(d_x) \wedge t_{tests}(d_x)) \wedge t_{in}(d_x)) \wedge t_{closed}(d_x)) \\ & \wedge \exists E.t_{value}(d_x) \end{aligned}$$

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

This function translates one specific constraint component: Unique Lang Constraint Component. Keywords: `sh:uniqueLang`.

Definition:

$$t_{lang}(E, d_x) = uniqueLang(E)$$

6 Defining $t_{target}(d_x)$

This function translates the Targets. Keywords: `sh:targetNode`, `sh:targetClass`, `sh:targetSubjectsOf`, `sh:targetObjectsOf`.

Definition:

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