SHACL Advanced Features

W3C Working Group Note 08 June 2017

This version:
https://www.w3.org/TR/2017/NOTE-shacl-af-20170608/

Latest published version:
https://www.w3.org/TR/shacl-af/

Latest editor's draft:
https://w3c.github.io/data-shapes/shacl-af/

Editors:
Holger Knublauch, TopQuadrant, Inc.
Dean Allemang, Working Ontologist LLC.
Simon Steyskal, WU Vienna/Siemens AG

Abstract

This document describes advanced features of the Shapes Constraint Language (SHACL) [shacl] including features to define custom targets, annotation properties, user-defined functions, node expressions and rules. While many of these features rely on SPARQL, they also define extension points that can be used by other implementation languages.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

Future revisions of this document may be produced by a SHACL W3C Community Group.

This document was published by the RDF Data Shapes Working Group as a First Public Working Group Note. Comments regarding this document are welcome. Please send them to public-rdf-shapes@w3.org (subscribe, archives).

Publication as a Working Group Note does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 March 2017 W3C Process Document.

Table of Contents

Document Conventions

Some examples in this document use Turtle [turtle]. The reader is expected to be familiar with SHACL [shacl] and SPARQL [sparql11-query].

Within this document, the following namespace prefix bindings are used:

Prefix	Namespace
`rdf:`	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
`rdfs:`	`http://www.w3.org/2000/01/rdf-schema#`
`sh:`	`http://www.w3.org/ns/shacl#`
`xsd:`	`http://www.w3.org/2001/XMLSchema#`
`ex:`	`http://example.com/ns#`

Throughout the document, color-coded boxes containing RDF graphs in Turtle will appear. These fragments of Turtle documents use the prefix bindings given above.

Example shapes graph

```
# This box represents a shapes graph
<s> <p> <o> .
```

```
// This box contains JavaScript code
```

Example data graph

```
# This box represents a data graph.
```

Example validation results

```
# This box represents an output results graph
```

Formal definitions appear in blue boxes:

TEXTUAL DEFINITIONS

```
# This box contains textual definitions.
```

Grey boxes such as this include syntax rules that apply to the shapes graph.

`true` denotes the RDF term `"true"^^xsd:boolean`. `false` denotes the RDF term `"false"^^xsd:boolean`.

Terminology

The terminology used throughout this document is consistent with the definitions in the main SHACL [shacl] specification, which references terms from RDF [rdf11-concepts]. This includes the terms ***binding***, ***blank node***, ***conformance***, ***constraint***, ***constraint component***, ***data graph***, ***datatype***, ***failure***, ***focus node***, ***RDF graph***, ***ill-formed***, ***IRI***, ***literal***, ***local name***, ***member***, ***node***, ***node shape***, ***object***, ***parameter***, ***pre-binding***, ***predicate***, ***property path***, ***property shape***, ***RDF term***, ***SHACL instance***, ***SHACL list***, ***SHACL subclass***, ***shape***, ***shapes graph***, ***solution***, ***subject***, ***target***, ***triple***, ***validation***, ***validation report***, ***validation result***, ***validator***, ***value***, ***value node***.

1. Introduction

The SHACL specification [shacl] is divided into SHACL Core and SHACL-SPARQL:

- **SHACL Core** consists of frequently needed features for the representation of shapes, constraints and targets.
- **SHACL-SPARQL** consists of all features of SHACL Core plus the expressive power of SPARQL-based constraints and an extension mechanism to declare new constraint components.

This document extends the functionality of SHACL by defining RDF vocabularies to cover the following features:

- **Custom Targets** - add flexibility to selecting the focus nodes of shapes
- **Annotation Properties** - can create extra values in validation reports
- **SHACL Functions** - encapsulate complex operations into reusable building blocks
- **Node Expressions** - describe how to derive sets of nodes
- **Expression Constraints** - enable constraint checks based on node expressions
- **SHACL Rules** - enable deriving new triples from existing ones

Taken together or individually, these features greatly extend the application scenarios of SHACL, and SHACL-SPARQL in particular.

2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words must and should are to be interpreted as described in [RFC2119].

Some of the features presented here (including node expressions, expression constraints and triple rules) do not necessarily require a SPARQL processor and could be used as extensions of pure SHACL Core implementations. Other features (including custom target types, SHACL functions, and general SHACL rules) define extension mechanisms that can also be used with other languages than SPARQL, such as JavaScript (as defined by the SHACL-JS document [shacl-js]).

A SHACL-SPARQL processor that also supports all features defined in this document and is called an ***Advanced SHACL-SPARQL*** processor.

3. Custom Targets

In general, targets define a mechanism that is used by SHACL engines to determine the focus nodes that should be validated against a given shape. SHACL Core [shacl] defines a fixed set of Core targets by means of properties such as `sh:targetClass`. These Core targets were designed to cover a large number of use cases while retaining a simple declarative data model. However, in some use cases, these Core targets are not sufficient. For example it is impossible to state that a shape should apply only to a subset of instances of a class, e.g. persons born in the USA. Neither is it possible to state that a shape should apply to all subjects in a graph, or to nodes selected by completely different, application-specific mechanisms.

This section defines richer mechanisms to define targets, called ***custom targets***. Custom targets are the values of the property `sh:target` in the shapes graph. The property `sh:target` has a similar status as, for example, `sh:targetClass`, and all subjects of `sh:target` triples are also shapes.

> The values of `sh:target` at a shape are IRIs or blank nodes.

A SHACL engine that supports custom targets uses the values of the custom target node to compute the target nodes for the associated shape. The algorithm that is used for this computation depends on the `rdf:type` of the custom target. The following sub-sections define two such algorithms:

- SPARQL-based targets
- SPARQL-based target types

However, other types of targets can be supported by other extension languages such as JavaScript. The class `sh:Target` is the recommended base class for such extensions.

The behavior of a SHACL engine that is unable to handle a given custom target is left undefined. SHACL Core processors do not even need to be aware of the existence of the `sh:target` property. Engines that are aware of this property and cannot handle a given custom target should at least report a warning.

3.1 SPARQL-based Targets

Custom targets that are SHACL instances of `sh:SPARQLTarget` are called ***SPARQL-based targets***.

> SPARQL-based targets have exactly one value for the property `sh:select`. SPARQL-based targets may have values for the property `sh:prefixes` and these values are IRIs or blank nodes. Using the values of `sh:prefixes` as defined by 5.2.1 Prefix Declarations for SPARQL Queries, the values of `sh:select` must be valid SPARQL 1.1 SELECT queries with a single result variable `this`. SPARQL-based targets have at most one value for the property `sh:ask`.

The following example declares a well-formed SPARQL-based target that produces all persons born in the USA:

Example shapes graph

```
ex:
   sh:declare [
      sh:prefix "ex" ;
```

```
      sh:namespace <http://example.com/ns#> ;
   ] .

ex:USCitizenShape
   a sh:NodeShape ;
   sh:target [
      a sh:SPARQLTarget ;
      sh:prefixes ex: ;
      sh:select """
         SELECT ?this
         WHERE {
            ?this a ex:Person .
            ?this ex:bornIn ex:USA .
         }
         """ ;
   ] ;
   ...
```

> TEXTUAL DEFINITION
>
> Let `Q` be the SPARQL SELECT query derived from the values of `sh:select` and `sh:prefixes` of the SPARQL-based target `T`. The target nodes of `T` are the bindings of the variable `this` returned by `Q` against the data graph.

While the SELECT queries can be used to identify all focus nodes for a given shape, SHACL processors sometimes also need to compute the inverse direction and find all shapes for which a given node needs to be validated against. For this reason, the following semantic restriction is recommended for SELECT queries used in SPARQL-based targets. Informally, SHACL Full processors should be able to derive an equivalent ASK query from the SELECT query, pre-bind the potential focus node, and check whether the potential focus node needs to be validated against the shape that has the given target. Formally, let `A` be a SPARQL ASK query that is produced by replacing the SelectClause with `ASK` in the outermost SELECT query. Let `rs` be the set of RDF terms returned as bindings for the variable `this` in the solutions of the SELECT query. Then `A` returns `true` if and only if the variable `this` is pre-bound with a value from `rs`. If the SELECT query of a SPARQL-based target does not fulfill this requirement, it needs to be accompanied by a SPARQL ASK query as the value for `sh:ask`. A SHACL engine can then determine whether a given shape applies to a given node by executing the ASK query with the variable `this` pre-bound to the node. If the ASK query evaluates to `true` then the node is in the target of the shape.

3.2 SPARQL-based Target Types

In some cases it would be too repetitive to declare SPARQL-based targets with similar SPARQL queries that only differ in a few aspects. SHACL-SPARQL defines a mechanism for user-defined constraint components, allowing users to reuse the same SPARQL query in a parameterized form. The SPARQL-based target types introduced in this section follow a similar design.

The class `sh:TargetType` can be used to declare high-level vocabularies for targets in a shapes graph. The class `sh:SPARQLTargetType` is declared as `rdfs:subClassOf sh:TargetType` for ***SPARQL-based target types***. Other extension languages may define alternative execution instructions for target types with the same IRI, making them potentially more platform independent than pure SPARQL-based targets. Instances of the class `sh:SPARQLTargetType` specify a SPARQL SELECT query via the property `sh:select`, and this query has to fulfill the same syntactic and semantic rules as SPARQL-based targets.

Similar to SPARQL-based constraint components, such targets take parameters and the parameter values become pre-bound variables in the associated SPARQL queries. The parameter values of such targets cannot not be blank nodes, and the same target cannot have more than one value per parameter. A target that lacks a value for a non-optional parameter is ignored, producing no target nodes. Similar to SPARQL-based constraint components, target types may also have values for the property `sh:labelTemplate`.

The following example declares a new SPARQL-based target type that takes one parameter `ex:country` that gets mapped into the variable `country` in the corresponding SPARQL query to determine the resulting target nodes.

Example shapes graph

```
ex:PeopleBornInCountryTarget
   a sh:SPARQLTargetType ;
   rdfs:subClassOf sh:Target ;
   sh:labelTemplate "All persons born in {$country}" ;
   sh:parameter [
      sh:path ex:country ;
      sh:description "The country that the focus nodes are 'born' in." ;
      sh:class ex:Country ;
      sh:nodeKind sh:IRI ;
   ] ;
   sh:prefixes ex: ;
   sh:select """
      SELECT ?this
      WHERE {
         ?this a ex:Person .
         ?this ex:bornIn $country .
      }
      """ .
```

Once such a target type has been defined in a shapes graph, it can be used by multiple shapes:

Example shapes graph

```
ex:GermanCitizenShape
   a sh:NodeShape ;
   sh:target [
      a ex:BornInCountryTarget ;
      ex:country ex:Germany ;
   ] ;
   ...

ex:USCitizenShape
   a sh:NodeShape ;
   sh:target [
      a ex:BornInCountryTarget ;
```

```
        ex:country ex:USA ;
    ] ;
    ...
```

The set of focus nodes produced by such a target type consists of all bindings of the variable `this` in the result set, when the SPARQL SELECT query has been executed with the pre-bound parameter values.

4. Annotation Properties

This section extends the general mechanism from SHACL-SPARQL [shacl] to produce validation reports as a result of the validation.

Implementations that support this feature make it possible to inject so-called ***annotation properties*** into the validation result nodes created for each solution produced by the SELECT queries of a SPARQL-based constraint or constraint component. Any such annotation property needs to be declared via a value of `sh:resultAnnotation` at the subject of the `sh:select` or `sh:ask` triple.

> The values of `sh:resultAnnotation` are called ***result annotations*** and are either IRIs or blank nodes.

Result annotations have the following properties:

Property	Summary and Syntax Rules
`sh:annotationProperty`	The property that shall be set. Each result annotation has exactly one value for the property `sh:annotationProperty` and this value is an IRI.
`sh:annotationVarName`	The name of the SPARQL variable to take the annotation values from. Each result annotation has at most 1 value for the property `sh:annotationVarName` and this value is literal with datatype `xsd:string`.
`sh:annotationValue`	Constant RDF terms that shall be used as default values.

For each solution of a SELECT result set, a SHACL processor that supports annotations walks through the declared result annotations. The mapping from result annotations to SPARQL variables uses the following rules:

1. Use the value of the property `sh:annotationVarName`
2. If no such value exists, use the local name of the value of `sh:annotationProperty` as the variable name

If a variable name could be determined, then the SHACL processor copies the binding for the given variable as a value for the property specified using `sh:annotationProperty` into the validation result that is being produced for the current solution. If the variable has no binding in the result set solution, then the values of `sh:annotationValue` is used, if present.

Here is an example illustrating the use of result annotations.

Example shapes graph

```
ex:AnnotationExample
   a sh:NodeShape ;
   sh:targetNode ex:ExampleResource ;
   sh:sparql [   # _:b1
      sh:resultAnnotation [
         sh:annotationProperty ex:time ;
         sh:annotationVarName "time" ;
      ] ;
      sh:select """
         SELECT $this ?message ?time
         WHERE {
            BIND (CONCAT("The ", "message.") AS ?message) .
            BIND (NOW() AS ?time) .
         }
         """ ;
   ] .
```

Validation produces the following validation report:

Example validation results

```
[  a sh:ValidationReport ;
   sh:conforms false ;
   sh:result [
      a sh:ValidationResult ;
      sh:focusNode ex:ExampleResource ;
      sh:resultMessage "The message." ;
      sh:resultSeverity sh:Violation ;
      sh:sourceConstraint _:b1 ;
      sh:sourceConstraintComponent sh:SPARQLConstraintComponent ;
      sh:sourceShape ex:AnnotationExample ;
      ex:time "2015-03-27T10:58:00"^^xsd:dateTime ;  # Example
   ]
] .
```

5. SHACL Functions

SHACL functions declare operations that produce an RDF term based on zero or more parameters and a data graph. Each SHACL function has an IRI. The actual execution logic (or algorithm) of a SHACL function can be declared in a variety of execution languages. This document defines one specific kind of SHACL functions, the SPARQL-based functions. JavaScript-based Functions are defined in the separate SHACL-JS document [shacl-js]. The same function IRI can potentially be executed on a multitude of platforms, if it declares execution instructions for these platforms.

SHACL functions can be called within FILTER or BIND clauses and similar features of SPARQL queries. SHACL functions can also be used declaratively in frameworks such as the SHACL node expressions which are used in SHACL rules. In those scenarios they may be used to perform data transformations such as string concatenation.

5.1 An Example SHACL Function

The following example illustrates the declaration of a SHACL function based on a simple mathematical SPARQL query.

Example shapes graph

```
ex:multiply
   a sh:SPARQLFunction ;
   rdfs:comment "Multiplies its two arguments $op1 and $op2." ;
   sh:parameter [
      sh:path ex:op1 ;
      sh:datatype xsd:integer ;
      sh:description "The first operand" ;
   ] ;
   sh:parameter [
      sh:path ex:op2 ;
      sh:datatype xsd:integer ;
      sh:description "The second operand" ;
   ] ;
   sh:returnType xsd:integer ;
   sh:select """
      SELECT ($op1 * $op2 AS ?result)
      WHERE {
      }
      """ .
```

Using the declaration above, SPARQL engines that support SHACL functions install a new SPARQL function based on the SPARQL 1.1 Extensible Value Testing mechanism. Such engines are then able to handle expressions such as `ex:multiply(7, 8)`, producing `56`, as illustrated in the following SPARQL query.

Example

```
SELECT ?subject ?area
WHERE {
   ?subject ex:width ?width .
   ?subject ex:height ?height .
   BIND (ex:multiply(?width, ?height) AS ?area) .
}
```

The following sections introduce the general properties that such functions may have, before the specific characteristics of SPARQL-based functions are defined.

5.2 Function Parameters

The parameters of a SHACL function are declared using the property `sh:parameter`. This corresponds closely to the parameter declarations of SPARQL-based constraint components, and the same syntax rules apply.

Parameters are ordered, corresponding to the notation of function calls in SPARQL such as `ex:exampleFunction(?param1, ?param2)`. The ordering of function parameters is determined as follows:

- If any of the parameters have a value for `sh:order` then all of them are ordered in ascending order by the parameters' numeric values of `sh:order`, using `0` as default value if unspecified.
- If none of the parameters have a value for `sh:order` then all of them are ordered in ascending order of the local names of their declared `sh:path` values.

Each parameter may have its property `sh:optional` set to `true` to indicate that the parameter is not mandatory. If a function gets invoked without all its mandatory parameters then it returns no result node (an error in SPARQL, producing unbound in a BIND statement).

5.3 sh:returnType

A function may declare a single ***return type*** via `sh:returnType`.

> A function has at most one value for `sh:returnType`. The values of `sh:returnType` are IRIs.

The return type may serve for documentation purposes only. However, in some execution languages such as JavaScript, the declared `sh:returnType` may inform a processor how to cast a native value into an RDF term.

5.4 SPARQL-based Functions

SHACL instances of `sh:SPARQLFunction` that are IRIs in a shapes graph are called ***SPARQL-based functions***.

> SPARQL-based functions have exactly one value for either `sh:ask` or `sh:select`. The values of these properties are strings that can be parsed into SPARQL queries of type ASK (for `sh:ask`) or SELECT (for `sh:select`) using the SHACL-SPARQL prefix declaration mechanism. SELECT queries return exactly one result variable and do not use the `SELECT *` syntax.

When the function is executed, the SPARQL processor needs to pre-bind variables based on the provided arguments of the function call. In the SHACL functions example above, the value for the parameter declared as `ex:op1` is pre-bound to the SPARQL variable `$op1`, etc. For ASK queries, the function's return value is the result of the ASK query execution, i.e. `true` or `false`. For SELECT queries, the function's return value is the binding of the (single) result variable of the first solution in the result set. Since all other bindings will be ignored, such SELECT queries should only return at most one solution. If the result variable is unbound, then the function generates a SPARQL error.

6. Node Expressions

This section defines a feature called ***node expressions***. Node expressions are declared as RDF nodes in a shapes graph and instruct a SHACL engine how to compute a set of nodes for a given focus node. Each node expression has one of the following types, each of which is defined together with its evaluation semantics in the following sub-sections.

Node Expression Type	Syntax (Informative)	Summary
Focus Node Expression	`sh:this`	The set consisting of the current focus node.

Constant Term Expression	Any IRI or literal except `sh:this`	The set consisting of the given term.
Function Expression	Blank node with a list-valued triple	The results of evaluating a given SHACL Function.
Path Expression	Blank node with `sh:path`	The values of a given property path.
Filter Shape Expression	Blank node with `sh:filterShape` and `sh:nodes`	The sub-set of the input nodes that conform to a given shape.
Intersection Expression	Blank node with `sh:intersection`	The intersection of two or more input sets.
Union Expression	Blank node with `sh:union`	The union set of two or more input sets.

The basic idea of these expressions is that they can be used to derive a set of RDF nodes from a given focus node, for example the set of all values of a given property of the focus node. Some of these expressions can be nested, i.e. they use the output of another expression as their input, leading to evaluation chains and trees.

The following example declares a node expression that produces the display labels of all values of the property `ex:customer` that conform to a given shape `ex:GoodCustomerShape`. The assumption here is that there is a SHACL function `ex:displayLabel` which declares a single parameter.

Example shapes graph

```
[
   ex:displayLabel ( [
      sh:filterShape ex:GoodCustomerShape ;
      sh:nodes [ sh:path ex:customer ] ;
   ] )
] .
```

To evaluate this example, an engine gets all values of `ex:customer` of the focus node, then filters them according to the shape `ex:GoodCustomerShape` and repeatedly calls the SHACL function `ex:displayLabel` with all values that pass the filter shape as arguments.

Important use cases of such expressions are expression constraints and SHACL rules, yet the basic functionality and vocabulary may find many other application areas.

Each of the following sub-sections defines a node expression type with its syntax rules and evaluation semantics based on a mapping operation `Eval($expr, $this)` where the first argument `$expr` is the given expression, `$this` is the current focus node and which produces a set of RDF nodes.

> A `node expression` cannot recursively have itself as a "nested" node expression, e.g. as value of `sh:nodes`.

6.1 Focus Node Expressions

The IRI `sh:this` is the (only) node declaring a ***focus node expression***.

> EVALUATION OF FOCUS NODE EXPRESSIONS

For the focus node expression `sh:this`, `Eval(sh:this, $this)` produces the set `{ $this }`.

6.2 Constant Term Expressions

Any literal or IRI except `sh:this` declares a ***constant term expression***.

EVALUATION OF CONSTANT TERM EXPRESSIONS

For the constant term expression `$expr`, `Eval($expr, $this)` produces the set `{ $expr }`.

6.3 Filter Shape Expressions

A ***filter shape expression*** is a blank node with exactly one value for `sh:filterShape` (which is a well-formed shape) and exactly one value for `sh:nodes` (which is a well-formed node expression).

EVALUATION OF FILTER SHAPE EXPRESSIONS

For the filter shape expression `$expr` with `S` being the shape that is the value of `sh:filterShape` and `N` being the node expression that is the value of `sh:nodes`, `Eval($expr, $this)` produces the set of nodes for each node `n` produced by evaluating `N` where `n` conforms to `S`.

6.4 Function Expressions

A ***function expression*** is a blank node that does not fulfill any of the syntax rules of the other node expression types and which is the subject of exactly one triple `T` where the object is a well-formed SHACL list, and each member of that list is a well-formed node expression.

EVALUATION OF FUNCTION EXPRESSIONS

For the function expression `$expr`, `Eval($expr, $this)` produces the set of nodes returned by evaluating the SHACL function specified as predicate of the triple `T` mentioned above. The arguments of the function call(s) are based on the results of the node expressions listed in the object list of `T` so that the first list member is used for the first argument, etc. This is done for all combinations of nodes produced by the node expression. If one of the node expressions produces the empty set and the corresponding function parameter is non-optional, then the result is the empty set.

As illustrated in the following example, function expressions are comparable to SPARQL BIND clauses.

Example shapes graph

```
[  ex:concat (
      [ sh:path ex:firstName ]
      [ sh:path ex:lastName ]
```

Example SPARQL

```
{
   $this ex:firstName ?a .
   $this ex:lastName ?b .
```

```
    )
] .
```

```
    BIND (ex:concat(?a, ?b) AS ?result) .
}
```

6.5 Path Expressions

A ***path expression*** is a blank node with exactly one value of the property `sh:path` (which are well-formed property paths) and at most one value for `sh:nodes` (which is a well-formed node expression).

EVALUATION OF PATH EXPRESSIONS

For the path expression `$expr` that has the property path `P` as its value for `sh:path` and the node expression `N` as its value for `sh:nodes` (defaulting to the focus node expression if absent), `Eval($expr, $this)` produces the set of values of all nodes produced by `N` for the property path `P`.

As illustrated in the following examples, path expressions are comparable to SPARQL basic graph patterns.

Example shapes graph

```
[  sh:path ex:firstName ] .

[  sh:nodes ex:children ;
   sh:path rdfs:label ;
] .
```

Example SPARQL

```
{   $this ex:firstName ?result . }

{   $this ex:children ?a .
    ?a rdfs:label ?result .
}
```

6.6 Intersection Expressions

An ***intersection expression*** is a blank node with exactly one value for the property `sh:intersection` which is a well-formed SHACL list with at least two members (which are well-formed node expressions).

EVALUATION OF INTERSECTION EXPRESSIONS

For the intersection expression `$expr` that has the list `L` as its value for `sh:intersection`, `Eval($expr, $this)` produces the set of nodes that are in all of the result sets produced by the members of `L`.

6.7 Union Expressions

A ***union expression*** is a blank node with exactly one value for the property `sh:union` which is a well-formed SHACL list with at least two members (which are well-formed node expressions).

EVALUATION OF UNION EXPRESSIONS

> For the union expression `$expr` that has the list `L` as its value for `sh:union`, `Eval($expr, $this)` produces the set of nodes that are in any of the result sets produced by all of the members of `L`.

As illustrated in the following example, union expressions are comparable to SPARQL UNION clauses.

Example shapes graph

```
[  sh:union (
      [ sh:path ex:firstName ]
      [ sh:path ex:givenName ]
   )
] .
```

Example SPARQL

```
{
   $this ex:firstName ?result .
} UNION {
   $this ex:givenName ?result .
}
```

7. Expression Constraints

Based on node expressions, this section introduces a constraint component called ***expression constraints***. Expression constraints can be used in any shape to declare the condition that the node expression specified via `sh:expression` has `true` as its (only) result. In the evaluation of these node expressions is repeated for all value nodes of the shape as the focus node.

Constraint Component IRI: `sh:ExpressionConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:expression`	The node expression that must return `true`. The values of `sh:expression` at a shape must be well-formed node expressions.

> TEXTUAL DEFINITION
>
> For each value node `v` where `Eval(v, $expression)` returns a node set that is not equal to `{ true }` there is a validation result that has `v` as its `sh:value` and `$expression` as its `sh:sourceConstraint`. If the `$expression` has values for `sh:message` in the shapes graph then these values become the (only) values for `sh:resultMessage` in the validation result.

The remainder of this section is informative.

The following example assumes that there are SHACL functions `ex:concat`, `ex:strlen` and `ex:lessThan` and uses them to verify that the combined string length of `ex:firstName` and `ex:lastName` is less than 30.

Example shapes graph

```
ex:FilterExampleShape
   a sh:NodeShape ;
   sh:expression [
      ex:lessThan (
         [ ex:strlen (
```

```
            [ ex:concat ( [ sh:path ex:firstName] [ sh:path ex:lastName ] ) ] )
        ]
        30 );
    ] .
```

8. SHACL Rules

SHACL defines an RDF vocabulary to describe shapes - collections of constraints that apply to a set of nodes. Shapes can be associated with nodes using a flexible target mechanism, e.g. for all instances of a class. One focus area of SHACL is data validation. However, the same principles of describing data patterns in shapes can also be exploited for other purposes. SHACL rules build on SHACL to form a light-weight RDF vocabulary for the exchange of rules that can be used to derive inferred RDF triples from existing *asserted* triples.

The SHACL rules feature defined in this section includes a general framework using the properties such as `sh:rule` and `sh:condition`, plus an extension mechanism for specific rule types. This document defines two such rule types: Triple rules and SPARQL rules. Other documents, including SHACL JavaScript Extensions [shacl-js], can define additional types of rules.

8.1 Examples of SHACL Rules

This section is non-normative.

The following example illustrates the use of a triple rule that adds an `rdf:type` triple so that those SHACL instances of `ex:Rectangle` where the `ex:width` equals the `ex:height` are also marked to be instances of `ex:Square`. The rule applies only to well-formed rectangles that conform to the `ex:Rectangle` shape, e.g. by having exactly one width and height, both integers.

Example shapes graph

```
ex:Rectangle
   a rdfs:Class, sh:NodeShape ;
   rdfs:label "Rectangle" ;
   sh:property [
      sh:path ex:height ;
      sh:datatype xsd:integer ;
      sh:maxCount 1 ;
      sh:minCount 1 ;
      sh:name "height" ;
   ] ;
   sh:property [
      sh:path ex:width ;
      sh:datatype xsd:integer ;
      sh:maxCount 1 ;
      sh:minCount 1 ;
      sh:name "width" ;
   ] ;
   sh:rule [
```

```
      a sh:TripleRule ;
      sh:subject sh:this ;
      sh:predicate rdf:type ;
      sh:object ex:Square ;
      sh:condition ex:Rectangle ;
      sh:condition [
         sh:property [
            sh:path ex:width ;
            sh:equals ex:height ;
         ] ;
      ] ;
   ] .
```

Example data graph

```
ex:InvalidRectangle
   a ex:Rectangle .

ex:NonSquareRectangle
   a ex:Rectangle ;
   ex:height 2 ;
   ex:width 3 .

ex:SquareRectangle
   a ex:Rectangle ;
   ex:height 4 ;
   ex:width 4 .
```

For the data graph above, a SHACL rules engine will produce the following inferred triples:

```
ex:SquareRectangle rdf:type ex:Square .
```

No inferences will be made for `ex:NonSquareRectangle` because its width is not equal to its height. No inferences will be made for `ex:InvalidRectangle` because although it has equal width and height (namely none), it does not pass the `sh:condition` of being a well-formed rectangle.

The following example illustrates a simple use case of a SPARQL rule that applies to all instances of the class `ex:Rectangle` and computes the values of the `ex:area` property by multiplying the rectangle's width and height:

Example shapes graph

```
ex:RectangleShape
   a sh:NodeShape ;
   sh:targetClass ex:Rectangle ;
   sh:property [
      sh:path ex:width ;
      sh:datatype xsd:integer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
   ] ;
   sh:property [
      sh:path ex:height ;
```

```
      sh:path ex:height ;
      sh:datatype xsd:integer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
   ] .

ex:RectangleRulesShape
   a sh:NodeShape ;
   sh:targetClass ex:Rectangle ;
   sh:rule [
      a sh:SPARQLRule ;
      sh:prefixes ex: ;
      sh:construct """
         CONSTRUCT {
            $this ex:area ?area .
         }
         WHERE {
            $this ex:width ?width .
            $this ex:height ?height .
            BIND (?width * ?height AS ?area) .
         }
         """ ;
      sh:condition ex:RectangleShape ;    # Rule only applies to Rectangles that conform to ex:Re
   ] ;
.
```

An engine that is capable of executing such rules uses the target statements associated with the shapes in the shapes graph to determine which rules need to be executed on which target nodes. For those target nodes that conform to any condition shapes, it executes the provided CONSTRUCT queries to produce the inferred triples. During the execution of the query, the variable `this` has the current focus node as pre-bound variable. For the following data graph, the triples below would be produced.

Example data graph

```
ex:ExampleRectangle
   a ex:Rectangle ;
   ex:width 7 ;
   ex:height 8 .

ex:InvalidRectangle    # Lacks a value for ex:height, so sh:condition is not met
   a ex:Rectangle ;
   ex:width 7 .
```

Inferred triples:

```
ex:ExampleRectangle ex:area 56 .
```

The following variation produces the same results as the SPARQL rule, but uses a Triple rule. While not as expressive as CONSTRUCT-based rules, Triple rules are more declarative and may be executed on platforms that do not support SPARQL.

Example shapes graph

```
ex:RectangleRulesShape
   a sh:NodeShape ;
   sh:targetClass ex:Rectangle ;
   sh:rule [
      a sh:TripleRule ;
      sh:subject sh:this ;
      sh:predicate ex:area ;    # Computes the values of the ex:area property at the focus nodes
      sh:object [
         ex:multiply ( [ sh:path ex:width ] [ sh:path ex:height ] ) ;
      ] ;
      sh:condition ex:RectangleShape ;    # Rule only applies to Rectangles that conform to ex:Re
   ] .
```

8.2 General Syntax of SHACL Rules

The values of the property `sh:rule` at a shape are called ***SHACL rules***. SHACL has a flexible design in which multiple types of rules can be supported, including Triple rules and SPARQL rules. Each ***rule type*** is identified by an IRI that is used as `rdf:type` of rules. Each rule type also defines ***execution instructions*** that can be implemented by rule engines.

> Each SHACL rule has at least one `rdf:type` which is a IRI.

Rules can have multiple types, e.g. to provide instructions that work either in SPARQL or JavaScript, depending on the capabilities of the engine. The creator of such rules needs to make sure that such rules have consistent semantics. Rule `R` has rule type `T` if `R` is a SHACL instance of `T`.

All rules may have the properties defined in the rest of this section.

8.2.1 sh:condition

A rule may have values for the property `sh:condition` to specify shapes that the focus nodes must conform to before the rule gets executed.

> The values of `sh:condition` at a rule must be well-formed shapes.

8.2.2 sh:order

Rules and shapes may specify its relative ***execution order*** as defined in this section.

> Each rule or shape may have at most one value for the property `sh:order`. The values of `sh:order` at rules and shapes are literals with a *numeric* datatype such as `xsd:decimal`.

If unspecified, then the default execution order is `0`. These values are used by a rules engine to determine the order of rules. When the rules associated with a shape are executed, rules with larger values will be executed after those with smaller values.

Example shapes graph

```
ex:RuleOrderExampleShape
   a sh:NodeShape ;
   sh:targetClass ex:Person ;
   sh:rule [
      a sh:SPARQLRule ;
      rdfs:label "Infer uncles, i.e. male siblings of the parents of $this" ;
      sh:prefixes ex: ;
      sh:order 1 ;   # Will be evaluated before 2
      sh:construct """
         CONSTRUCT {
            $this ex:uncle ?uncle .
         }
         WHERE {
            $this ex:parent ?parent .
            ?parent ex:sibling ?uncle .
            ?uncle ex:gender ex:male .
         }
         """
   ] ;
   sh:rule [
      a sh:SPARQLRule ;
      rdfs:label "Infer cousins, i.e. the children of the uncles" ;
      sh:prefixes ex: ;
      sh:order 2 ;
      sh:construct """
         CONSTRUCT {
            $this ex:cousin ?cousin .
         }
         WHERE {
            $this ex:uncle ?uncle .
            ?cousin ex:parent ?uncle .
         }
         """
   ] .
```

8.2.3 sh:deactivated

Rules may be *deactivated* by setting `sh:deactivated` to `true`. Deactivated rules are ignored by the rules engine.

> Each rule may have at most one value for the property `sh:deactivated`. The values of `sh:deactivated` are either of the `xsd:boolean` literals `true` or `false`.

8.3 The sh:Rules Entailment Regime

SHACL defines the property `sh:entailment` to link a shapes graph with *entailment regimes*. The IRI `sh:Rules` represents the ***SHACL rules entailment regime***. In the following example, the shapes graph indicates to a SHACL validation engine that the SHACL rules inside of the shapes graph need to be executed prior to starting the validation.

Example shapes graph

```
<http://example.org/my-shapes>
   a owl:Ontology ;
   sh:entailment sh:Rules .
```

Following the general policy for SHACL, validation engines that do *not* support the SHACL rules entailment regime must signal a failure if this triple is present. Validation engines that do support the SHACL rules entailment regime execute the rules following the rules execution instructions prior to performing the actual validation.

8.4 General Execution Instructions for SHACL Rules

A ***SHACL rules engine*** is a computer procedure that takes as input a data graph and a shapes graph and is capable of adding triples to the data graph. The new triples that are produced by a rules engine are called the ***inferred*** triples.

Note that, from a logical perspective, the data graph will be *modified* if triples get inferred. This means that rules can trigger after other triples have been inferred. However, in cases where the original data should not be modified, implementations may construct a logical data graph that has the original data as one subgraph and a dedicated inferences graph as another subgraph, and where the inferred triples get added to the inferences graph only.

In order to count as a SHACL rules engine, an implementation must be capable of inferring triples according to the following procedure (given in pseudo-code), or a different algorithm as long as the result is the same as specified. Note that this algorithm only covers a single "iteration" over all rules, without prescribing the behavior if the same rule needs to be applied multiple times after other rules have fired. The latter is left to future work.

```
for each shape S in the shapes graph, ordered by execution order {
   for each non-deactivated rule R in the shape, ordered by execution order {
      for each target node T of S that conforms to all conditions of R {
         execute R using T as focus node following the execution instructions of its rule types
      }
   }
}
```

The triples that are inferred by a rule do not *immediately* become part of the data graph, i.e. the triples produced by one rule can not always be queried by other rules. These policies reduce the likelihood of race conditions and better support parallel execution.

- If two shapes have the same execution order then their newly inferred triples are not visible to each other.
- If two rules have the same execution order then their newly inferred triples are not visible to each other.
- If the same rule is executed on multiple target nodes then the newly inferred triples are not visible to the other target nodes.

If a rules engine is not able to execute a given rule because it does not support any of the rule types of the rule, then it reports a failure.

At no time are inferred triples visible to the shapes graph, i.e. it is impossible for rules to modify the definitions of rules or shapes.

8.5 Triple Rules

This section defines a rule type called ***triple rules***, identified by the IRI `sh:TripleRule`. Triple rules have the following properties:

Property	Summary and Syntax Rules
`sh:subject`	The node expression used to compute the subjects of the triples. Each triple rule must have exactly one value of the property `sh:subject` (which must be a well-formed node expression).
`sh:predicate`	The node expression used to compute the predicates of the triples. Each triple rule must have exactly one value of the property `sh:predicate` (which must be a well-formed node expression).
`sh:object`	The node expression used to compute the objects of the triples. Each triple rule must have exactly one value of the property `sh:object` (which must be a well-formed node expression).

EXECUTION OF TRIPLE RULES

Let `S`, `P` and `O` be the sets of nodes produced by evaluating the node expressions that are the values of `sh:subject`, `sh:predicate` and `sh:object` respectively at the triple rule. For each combination of members `s` of `S`, `p` of `P` and `o` of `O`, infer a triple with subject `s`, predicate `p` and object `o`.

8.6 SPARQL Rules

This section defines a rule type called ***SPARQL rules***, identified by the IRI `sh:SPARQLRule`. SPARQL rules have the following properties:

Property	Summary and Syntax Rules
`sh:construct`	The SPARQL CONSTRUCT query. SPARQL rules must have exactly one value for the property `sh:construct`. The values of `sh:construct` are literals with datatype `xsd:string`.
`sh:prefixes`	The prefixes to use to turn the `sh:construct` into a SPARQL query. SPARQL rules may use the property `sh:prefixes` to declare a dependency on prefixes based on the mechanism defined in Prefix Declarations for SPARQL Queries from the SHACL specification [shacl]. This mechanism allows users to abbreviate URIs in the `sh:construct` strings.

EXECUTION OF SPARQL RULES

Let `Q` be the SPARQL CONSTRUCT query derived from the values of the properties `sh:construct` and `sh:prefixes` of the SPARQL rule in the shapes graph. For each focus node, execute the query `Q` pre-binding the

> variable `this` to the focus node, and infer the constructed triples.

A. Summary of Syntax Rules from this Document

This section enumerates all normative syntax rules from this document. This section is automatically generated from other parts of this spec and hyperlinks are provided back into the prose if the context of the rule in unclear. Nodes that violate these rules in a shapes graph are ill-formed.

Syntax Rule Id	Syntax Rule Text
target-nodeKind	The values of `sh:target` at a shape are IRIs or blank nodes.
SPARQLTarget-select-count	SPARQL-based targets have exactly one value for the property `sh:select`.
SPARQLTarget-prefixes-nodeKind	SPARQL-based targets may have values for the property `sh:prefixes` and these values are IRIs or blank nodes.
SPARQLTarget-select-sparql	Using the values of `sh:prefixes` as defined by 5.2.1 Prefix Declarations for SPARQL Queries, the values of `sh:select` must be valid SPARQL 1.1 SELECT queries with a single result variable `this`.
SPARQLTarget-ask-count	SPARQL-based targets have at most one value for the property `sh:ask`.
resultAnnotation-nodeKind	The values of `sh:resultAnnotation` are called result annotations and are either IRIs or blank nodes
annotationProperty	Each result annotation has exactly one value for the property `sh:annotationProperty` and this value is an IRI.
annotationVarName	Each result annotation has at most 1 value for the property `sh:annotationVarName` and this value is literal with datatype `xsd:string`.
returnType-maxCount	A function has at most one value for `sh:returnType`.
returnType-nodeKind	The values of `sh:returnType` are IRIs.
SPARQLFunction-query	SPARQL-based functions have exactly one value for either `sh:ask` or `sh:select`. The values of these properties are strings that can be parsed into SPARQL queries of type ASK (for `sh:ask`) or SELECT (for `sh:select`) using the SHACL-SPARQL prefix declaration mechanism. SELECT queries return exactly one result variable and do not use the `SELECT *` syntax.
node-expressions-recursion	A `node expression` cannot recursively have itself as a "nested" node expression, e.g. as value of `sh:nodes`.
FilterShapeExpression	A filter shape expression is a blank node with exactly one value for `sh:filterShape` (which is a well-formed shape) and exactly one value for `sh:nodes` (which is a well-formed node expression).
FunctionExpression	A function expression is a blank node that does not fulfill any of the syntax rules of the other node expression types and which is the subject of exactly

	one triple `T` where the object is a well-formed SHACL list, and each member of that list is a well-formed node expression.
PathExpression	A path expression is a blank node with exactly one value of the property `sh:path` (which are well-formed property paths) and at most one value for `sh:nodes` (which is a well-formed node expression).
intersection	An intersection expression is a blank node with exactly one value for the property `sh:intersection` which is a well-formed SHACL list with at least two members (which are well-formed node expressions).
union	A union expression is a blank node with exactly one value for the property `sh:union` which is a well-formed SHACL list with at least two members (which are well-formed node expressions).
expression-scope	The values of `sh:expression` at a shape must be well-formed node expressions.
rule-type	Each SHACL rule has at least one `rdf:type` which is a IRI.
condition-node	The values of `sh:condition` at a rule must be well-formed shapes.
rule-order-maxCount	Each rule or shape may have at most one value for the property `sh:order`.
rule-order-datatype	The values of `sh:order` at rules and shapes are literals with a *numeric* datatype such as `xsd:decimal`.
deactivated-maxCount	Each rule may have at most one value for the property `sh:deactivated`.
deactivated-in	The values of `sh:deactivated` are either of the `xsd:boolean` literals `true` or `false`.
TripleRule-subject	Each triple rule must have exactly one value of the property `sh:subject` (which must be a well-formed node expression).
TripleRule-predicate	Each triple rule must have exactly one value of the property `sh:predicate` (which must be a well-formed node expression).
TripleRule-object	Each triple rule must have exactly one value of the property `sh:object` (which must be a well-formed node expression).
construct-count	SPARQL rules must have exactly one value for the property `sh:construct`.
construct-datatype	The values of `sh:construct` are literals with datatype `xsd:string`.

B. Security and Privacy Considerations

This section is non-normative.

The features defined in this document share certain security and privacy considerations with those mentioned in [shacl]. The general advice is for users to only use trusted and controlled shape graphs.

C. Acknowledgements

This section is non-normative.

Many people contributed to this document, including members of the RDF Data Shapes Working Group. The sections 3. Custom Targets, 4. Annotation Properties and 5. SHACL Functions had been part of earlier drafts of the main SHACL specification [shacl] but were moved out in part due to time constraints in the Working Group. Dimitris Kontokostas was the main contributor to the 4. Annotation Properties section.

D. References

D.1 Normative references

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[rdf11-concepts]
RDF 1.1 Concepts and Abstract Syntax. Richard Cyganiak; David Wood; Markus Lanthaler. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/rdf11-concepts/

[shacl]
Shapes Constraint Language (SHACL). Holger Knublauch; Dimitris Kontokostas. W3C. 11 April 2017. W3C Candidate Recommendation. URL: https://www.w3.org/TR/shacl/

[sparql11-query]
SPARQL 1.1 Query Language. Steven Harris; Andy Seaborne. W3C. 21 March 2013. W3C Recommendation. URL: https://www.w3.org/TR/sparql11-query/

[turtle]
RDF 1.1 Turtle. Eric Prud'hommeaux; Gavin Carothers. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/turtle/

D.2 Informative references

[shacl-js]
SHACL JavaScript Extensions. Holger Knublauch; Pano Maria. W3C. W3C Working Group Note. URL: https://www.w3.org/TR/shacl-js/

↑