sheval: An RDF data shapes evaluation tool and test-suite for recursive shapes

Shqiponja Ahmetaj[1], Iovka Boneva[2], Jan Hidders[3], Maxime Jakubowski[1], Jose-Emilio Labra-Gayo[4,*], Wim Martens[5], Filip Murlak[6], Cem Okulmus[7], Ognjen Savković[8], Mantas Šimkus[1] and Dominik Tomaszuk[9]

[1]*TU Wien, Austria*
[2]*Univ. Lille, F-59000 Lille, France*
[3]*Birkbeck, University of London, UK*
[4]*University of Oviedo, Spain*
[5]*University of Bayreuth, Germany*
[6]*University of Warsaw, Poland*
[7]*Paderborn University, Germany*
[8]*Free University of Bolzano, Italy*
[9]*University of Białystok, Poland*

Abstract

Two different languages have been developed to validate RDF data based on the concept of a shape: ShEx and SHACL. In each language it is possible to define a shape that refers to itself, which is called a recursive shape. While in the case of ShEx, the semantics of recursive shapes is well defined and is part of the specification, in the case of SHACL, the semantics of recursive shapes is left to the implementation of the different SHACL engines. Consequently, the different SHACL engines show different behaviours when confronted with recursive shapes. In this paper we present sheval: an evaluation framework consisting of a tool and a test suite that can be used to compare the behaviour of different shapes technologies when confronted with recursive definitions. The tool has been used to evaluate and understand the differences in the implementation of recursive shapes in ShEx and SHACL. It provides a framework for testing and comparing the behaviour of different shape engines, helping to identify inconsistencies and potential issues, and providing a basis for further research and development in the field of shape-based validation of RDF data.

Keywords

SHACL, ShEx, RDF, Validation, Recursion, Shape languages, Evaluation, Benchmarking, Tool, Framework

1. Introduction

The need for validating RDF data has led to the development of two shape-based languages: ShEx and SHACL. While both languages share the goal of validating RDF data, they differ in their approach and implementation.

ShEx [1] was created as a concise and human-readable language, inspired by regular expressions and the idea of defining a ShEx schema as a set of shapes that describe a kind of grammar for RDF data. ShEx fully embraced the concept of recursive shapes, allowing for the definition of shapes that can refer to themselves [2]. In order to accommodate the combination of recursive shapes with negation, the semantics of ShEx was defined in terms of stratified negation which is a well-defined and widely accepted approach for handling recursion and negation in logic programming [2].

SHACL, on the other hand, was designed to provide a comprehensive framework for validating RDF data as a set of constraints that can be applied to RDF graphs. While SHACL also allows for the definition of recursive shapes, the semantics of recursive shapes is left to the implementation of the different SHACL engines. As a result, different SHACL engines may exhibit different behaviours in the

2nd International Workshop on Data Management for Knowledge Graphs (DMKG 2026)
*Corresponding author.
https://labra.weso.es/ (J. Labra-Gayo)
0000-0001-8907-5348 (J. Labra-Gayo); 0000-0002-7742-0439 (C. Okulmus); 0000-0003-1806-067X (D. Tomaszuk)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

```
prefix :   <http://example.org/>

:a :name "Alice" ;
    :knows :c .
:b :name "Bob" ;
   :knows :c .
:c :name "Carol" ;
   :knows :a     .
```

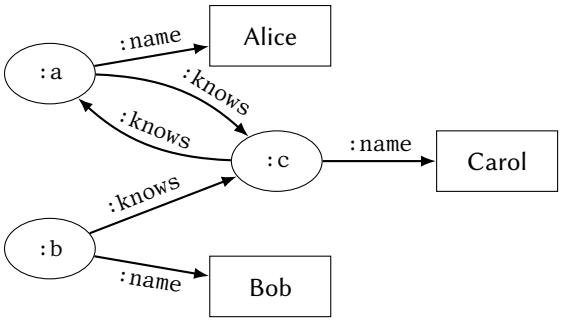

Figure 1: Example RDF graph representing a simple social network

presence of recursive shapes, leading to inconsistencies and potential issues in the validation process. There have been several approaches to describe the SHACL semantics with recursive shapes, including the use of fixed-point semantics and stratified negation but none of them has been adopted as part of the W3C recommendation. Although SHACL 1.2 specification is currently being developed, it does not yet provide a definitive solution to this issue[1].

A formal analysis of different notions of recursion for shape-based languages was done in [3]. In that paper, the sheval tool was developed as a framework for testing and comparing the behaviour of different shape engines, specifically addressing recursion. The purpose of sheval is to help practitioners and researchers identify inconsistencies between, and potential issues with, existing ShEx and SHACL validators when confronted with different test cases. In particular, we focus on the behaviour of these validators when dealing with recursive shapes and propose a test suite which covers different scenarios of recursion, including the combination of recursion with negation.

The main contribution of this paper is to present a description of the architecture of the sheval tool and an extended test suite for recursive shapes which includes control tests for non-recursive shapes and tests for the duality proposition between LFP and GFP as well as a more in-depth analysis of the results.

The SHEVAL tool is available as an open-source project on GitHub[2], including the source code, three test suites and the binaries of the engines employed in the comparison. The tool can be run in Docker to help reproducibility of the results.

2. Simple Shape Language and Recursive shapes

Definition 1. *Given a set of IRIs I, a set of blank-nodes B, and a set of literals L, an RDF graph G is a set of triples (s, p, o) where $s \in I \cup B$ is the subject, $p \in I$ is the predicate, and $o \in I \cup B \cup L$ is the object.*

Example 1. *Figure 1 shows an RDF graph in Turtle notation. It represents a simple social network with nodes :a, :b, and :c whose names are "Alice", "Bob" and "Carol", respectively, where :a knows :c, :b knows :c and :c knows :a. Figure 1 shows a graphical representation of this same RDF graph.*

Following [3], we define a simple shape language that can be used to describe the concept of recursive shapes.

Definition 2. *Given a set of shape names N, we define the set of shapes S as the set of all possible* shapes *φ defined by the following syntax, where $s \in N$ is a shape name, $p \in I$ is a predicate, and* cond *is a boolean condition over nodes.*

$$\varphi ::= \bot \mid \top \mid \text{cond} \mid s \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists p.\varphi \mid \forall p.\varphi.$$

[1] https://www.w3.org/TR/shacl12-core/#shapes-recursion
[2] https://github.com/cogsl/sheval

An informal semantics of the language features is defined as follows[3]*:*

- $\bot$ *represents a "semantically empty" shape, which is not satisfied by any node and* $\top$ *the universal shape, which is satisfied by all nodes.*
- cond *is satisfied if the node satisfies* cond. *In this paper will use only two conditions:* $\in$ *String which is satisfied if the node is a String literal, and* test(n) *which is satisfied if the node is* n.
- s *is a reference to another shape. A node* n *conforms to* s *when it satisfies the definition of* s. *Given that the language can have recursive definitions, the precise semantics in case of recursion will need shape assignments which will be introduced later.*
- $\neg\varphi$ *is the negation of a shape, satisfied by a node if it does not satisfy* φ.
- $\varphi_1 \vee \varphi_2$ *is satisfied if a node satisfies at least one of* φ_1 *or* φ_2.
- $\varphi_1 \wedge \varphi_2$ *is satisfied if a node satisfies both shapes.*
- $\exists p.\varphi$ *is satisfied by a node if there is an outgoing predicate* p *to a node that satisfies* φ.
- $\forall p.\varphi$ *is satisfied if all outgoing edges with predicate* p *lead to nodes that satisfy* φ.

A simple shape language catalog $C : N \to S$ *is a partial function mapping shape names to shapes. We assume that all shapes used in the image of the function are defined.*

Example 2. *We can define a simple shape language catalog* C *that describes the social network in Figure 1 as follows:*

$$C = \{\ user \mapsto person\ \wedge \exists\, name.(\in String)\ \wedge \exists\, knows.\top,\quad person \mapsto user\ \wedge \forall\, knows.person\ \}$$

The catalog defines two mutually recursive shapes: user and person. A node conforms to shape user if it conforms to person and has at least one property name whose value is a literal string and at least another property knows. A node conforms to the shape person if it also conforms to user and all values of property knows conform to person.

Definition 3. *A shape assignment* α *for a shapes catalog* C *and graph* G *is a relation* $\alpha \subseteq (I \cup B \cup L) \times N$ *that associates nodes in the graph with shape names in the schema. A shape assignment* α *is said to be* valid *if and only if for every* $(n, s) \in \alpha$, *the node* n *satisfies the shape* $C(s)$ *according to the semantics of the simple shape language.*

Shape assignments are necessary to indicate which nodes in the graph we want to validate against which shapes, in ShEx they are called shape maps, while in SHACL they can be defined with target declarations. Shape assignments can also be used to identify the result of the validation and during the validation with recursive shapes, they are used to define the semantics of the validation.

When dealing with recursive shapes, it is possible that there is more than one valid shape assignment for a given schema and graph.

Example 3. *Given the catalog* C *from example 2 and the graph* G *from example 1, notice that there is one possibility to assign the shape user to all nodes in the graph, another possibility would be to assign the shape user to nodes* $:a$ *and* $:c$, *and a third possibility would be to assign no shape to any node in the graph. Hence, we have three valid shape assignments* $\alpha_1, \alpha_2, \alpha_3$ *:*

$$\alpha_1 = \{(:a, user), (:b, user), (:c, user), (:a, person), (:b, person), (:c, person)\}$$
$$\alpha_2 = \{(:a, user), (:c, user), (:a, person), (:c, person)\}$$
$$\alpha_3 = \{\}$$

There are three common semantics for recursive shapes: least fixed point (LFP), greatest fixed point (GFP), and supported model semantics (SMS). LFP assumes that a node does not conform to a shape unless it satisfies the shape's constraints, in the previous example, LFP is α_3. GFP assumes that a node conforms to a shape unless it violates the shape's constraints so in the previous example, GFP is α_1.

[3]For the complete formal semantics, we refer to [3]

```
:User @:Person AND {
  :name  xsd:string    ;
  :name  .           * ;
  :knows .           + ;
}

:Person @:User AND {
  :knows @:Person    *
}
```

```
:User a sh:NodeShape ;
  sh:and (
      :Person
      [ sh:property
          [ sh:path :name ;
            sh:qualifiedValueShape [
              sh:datatype xsd:string
            ];
            sh:qualifiedMinCount 1
          ] ]
      [ sh:property
          [ sh:path :knows ;
            sh:minCount 1
          ] ] ) .

:Person a sh:NodeShape ;
    sh:and (
      :User
      [ sh:property
          [ sh:path :knows ;
            sh:node :Person ;
          ] ;
      ]) .
```

Figure 2: ShEx schema, SHACL shapes graph corresponding to example 2

The SMS looks for any self-supporting interpretation, not necessarily the least or the greatest one, and in the previous example, the SMS is the set $\{\alpha_1, \alpha_2, \alpha_3\}$. In the case of SMS, a validator is considered to use brave Supported Model Semantics (bSMS) if it checks that its result is any of the possible shape assignments, and it is considered cautious Supported Model Semantics (cSMS) if it checks all possible shape assignments.

For more details about LFP and GFP semantics, [3] presents some results that show that there is a significant fragment of ShEx with GFP and of SHACL with LFP that have the same expressive power, with possible translations between them.

3. ShEx, SHACL and recursive shapes

Although both ShEx and SHACL are shape-based languages for validating RDF data, they differ in their approach and implementation. In this section, we will discuss some of the key differences between ShEx and SHACL engines and how they handle recursive shapes.

Figure 2 presents both the ShEx schema and the equivalent SHACL shapes graph corresponding to Example 2. In the case of ShEx, we added a second declaration for :name to capture the semantics of the Simple Shape Language in which the property declarations are open. Similarly, in the case of SHACL, we use sh:qualifiedValueShape to capture the semantics of the Simple Shape Language, which would allow extra values for :name.

ShEx validators usually have three inputs, an RDF graph, a ShEx schema and a shape map that describes which nodes should be checked against which shapes, and they return a result map with the values that conform (or not) to the expected shapes. The shape maps are similar to shape assignments that indicate which nodes conform with which shapes.

Example 4. *A shape map to validate the nodes :a, :b and :c against the shape :User is :a@:User, :b@:User, :c@:User. Applying that shape map to the RDF graph in example 1, the basic result shape map obtained in ShEx is :a@:User, :b@:User, :c@:User, indicating that all three nodes conform to the shape*

:User (in practice, ShEx validators return the result using different syntaxes like JSON or a table with more information about the evidences for conformance/nonconformance).

In the previous example, the conformance is based on the fact that ShEx specification [4] which is based on [5] explicitly declares GFP semantics to evaluate recursive shapes.

SHACL engines have as input an RDF graph and a SHACL shapes graph, and they return a validation report that indicates if the RDF graph conforms to the SHACL shapes graph or if there are violations. SHACL shapes can also include target declarations which act like shape assignments. After running a SHACL validator with an RDF graph, a shapes graph and target declarations, it is expected to return a validation report which returns conforms=true if all the nodes selected by the target declarations conform to the expected shapes or conforms=false and a list violation errors if some nodes don't conform to their expected shapes. The SHACL recommendation does not specify a specific semantics for recursive shapes, so the behaviour of SHACL engines confronted with recursive shapes is not uniform.

Example 5. *If we run the previous example with the SHACL engine `TopQuadrant SHACL API` [6], the result obtained is the following validation report which indicates that it considers that the nodes :a and :b don't conform to the shape :Person:*

```
[ rdf:type      sh:ValidationReport;
  sh:conforms   false;
  sh:result [ a sh:ValidationResult;
    sh:focusNode :a;
    sh:resultMessage
      "Value must have all of the following shapes: :User, _:1";
    sh:resultSeverity sh:Violation;
    sh:sourceConstraintComponent sh:AndConstraintComponent;
    sh:sourceShape :Person;
    sh:value :a
  ];
  sh:result [ a sh:ValidationResult;
    sh:focusNode :b;
    sh:resultMessage
      "Value must have all of the following shapes: :User, _:2";
    sh:resultSeverity sh:Violation;
    sh:sourceConstraintComponent  sh:AndConstraintComponent;
    sh:sourceShape :Person;
    sh:value :b
   ]
  . . . # Similar for node :c
] .
```

By contrast, if we run the example with the Apache Jena SHACL engine [7] the result obtained is the following validation report:

```
[ rdf:type      sh:ValidationReport;
  sh:conforms   true;
] .
```

There are other possible behaviours: rudof [8] generates an error indicating that it doesn't support recursive shapes, pySHACL [9] generates a warning but continues the processing trying to validate and crashes with the message "Validation path too deep!" and SHACL-S [10] enters in an infinite loop without any warning.

Given that the SHACL 1.0 recommendation left handling of recursive shapes undefined, all the behaviours presented in the previous example are not breaking the recommendation. However, this

situation leads to inconsistencies and lack of interoperability between different SHACL implementations and users' expectations. In the next section we present a tool and a set of tests that can be used to compare the behaviour of different shapes technologies when confronted with recursive definitions.

4. sheval architecture

SHEVAL is a command-line framework for running the same shape-validation test suite against different *engines* that implement ShEx or SHACL. It also compares the obtained results against the expected semantics that a suite may prescribe. Its design follows a single guiding principle: everything that is *specific to one validation technology* (how to invoke it, how to parse its output, how it signals a crash) is isolated behind a small, uniform `Runner` interface, while everything that is *common to all of them* (test orchestration, result classification, semantics comparison, report generation) is implemented exactly once. Figure 3 gives an overview of the resulting module structure.

Test suites as data. A test suite is a directory described by a `manifest.yaml` file, listing the RDF/SHACL/ShEx source folders, the implementations to run per engine, and the test cases themselves. Each test case declares a data graph, a schema, and the *nodes* and *shapes* or *node–shape pairs* to validate, plus an optional `expected_results` block declaring the verdict prescribed by one or more semantics (the greatest/least fixed points, GFP/LFP, and the brave/cautious supported model semantics, bSMS/cSMS) as sets of pairs expected to pass or fail. Because this is pure data, adding a test case or a whole new suite never touches the code.

CLI and orchestration. The entry point exposes four subcommands: `test` runs a full suite; `shacl`/`shex` invoke a single technology directly on a data/schema pair; and `check` runs a technology's raw binary. For `test`, the *orchestrator* loads the manifest and, per test case, enriches the declared nodes/shapes/pairs with full IRIs and delegates to `graph_utils` to materialize each engine's expected input: a merged RDF graph with `sh:targetNode` declarations for SHACL, or a ShapeMap file for ShEx. It then dispatches these inputs to every requested technology and collects the results.

The Runner framework. Each technology is wrapped by a `Runner` subclass (e.g. `PyshaclRunner`, `JenaShaclRunner`, etc), grouped under a `SHACLRunner` or `ShExRunner` marker class. A concrete runner supplies a command-line *template* for its binary and implements two hooks: `execute`, which fills the template, invokes the binary through the shared `commands` module, and hands the output to `analysis` for parsing into a common `{conforms, successes, failures, message}` shape; and an optional `classify_error`, recognizing that engine's own failure signatures (e.g. rudof's "Dependency graph has cycles", pySHACL's "Validation path too deep!"), etc. The base class's `run` wraps every call in a common exception handler, so one technology crashing never aborts the suite.

Command execution and error classification. Command execution is provided by a `commands` module, which is in charge of spawning subprocesses: it runs a binary under a timeout and returns a `RunOutcome` with exit status, stderr and return code. `error_classification` then maps a failed run to a recognized failure mode — `Timeout`, `Crashed`, `CyclesDetected(Crashed/Conformant/NonConformant)`, `NonStratified(Crashed)` — first via the runner's own `classify_error` hook, then via generic, engine-agnostic crash signatures. This keeps the cross-cutting control flow in one place while each engine contributes only the patterns it actually produces.

Semantics matching and reporting. The `semantics` module compares an engine's reported successes/failures against every semantics declared in a test's `expected_results`, recording agreement with at least one (brave) or all (cautious) of the candidate models. The annotated results are finally handed to one or more exporters: plain YAML/CSV, or a LaTeX table like the one employed for Table 1.

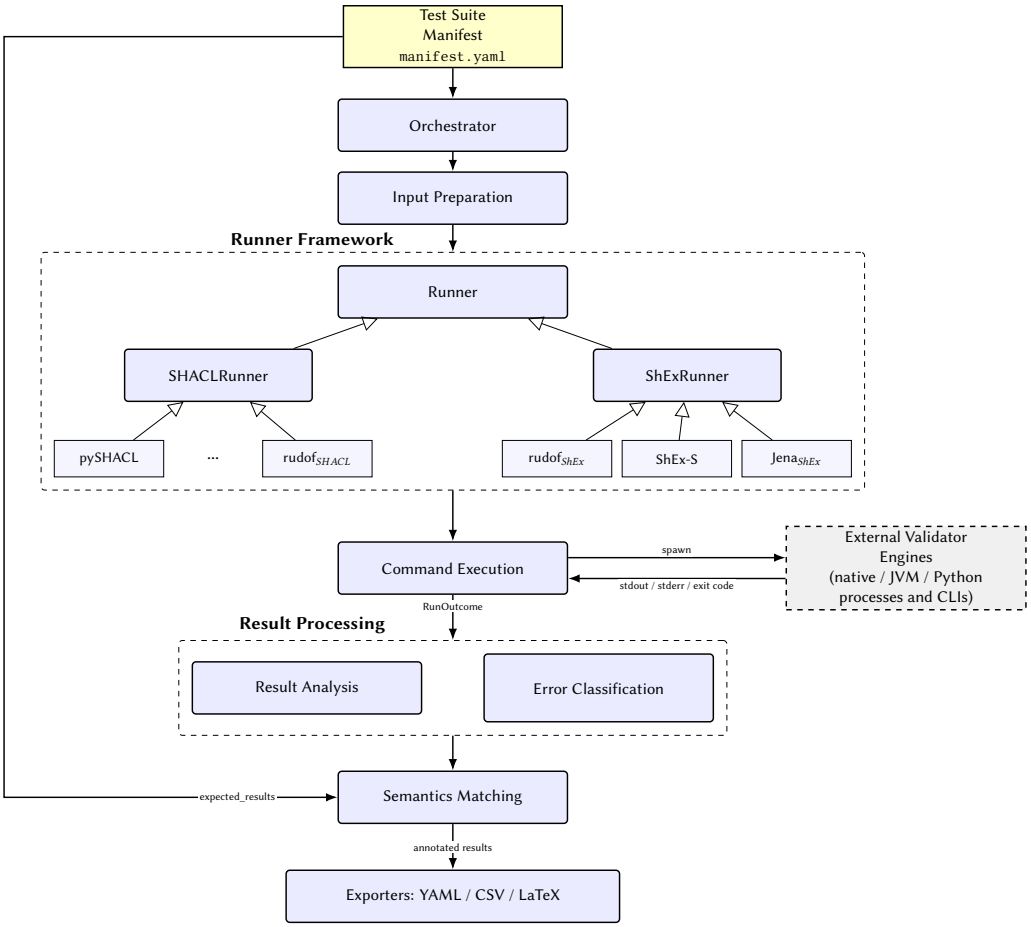

Figure 3: Main modules of SHEVAL: the YAML manifest drives the orchestrator, which prepares engine-specific inputs; runners invoke external validation engines whose output is parsed, classified and checked against the expected results and exported.

The repository includes a Docker-based setup with pinned dependencies and bundled validator binaries for pySHACL, SHACL-S, rudof (ShEx and SHACL), Apache Jena (ShEx and SHACL), SHACL-TQ and ShEx-S, enabling reproducible cross-engine comparisons[4]. Adding a new validator only requires to analyze and capture the template of the command required to run the validator and how it handles its output and potential warnings or errors, configure the necessary dependencies in Docker and implement a subclass of either ShExRunner or SHACLRunner.

5. Recursive shapes test suite

We developed a test suite to compare the behaviour of different shape engines when confronted with recursive shape definitions. The test suite aims to define simple tests that check the different semantic possibilities of recursive shapes. The test cases fall into two categories: separation and feature tests. Separation tests attempt to distinguish whether an engine uses LFP, GFP, brave SMS, or cautious SMS. Feature tests check some specific features of the engines. Each test is defined as $t_i = \langle G, C, sel, \alpha_{LFP}, \alpha_{GFP}, \alpha_1, \ldots, \alpha_n \rangle$ where G is an RDF graph, C is a shape catalog, and sel is a selector, where α_{LFP} is the expected shape assignment with least fixed-point semantics, α_{GFP} with greatest fixed-point semantics, and $\alpha_1, \ldots, \alpha_n$ are the remaining expected shape assignments in case of multiple valid interpretations in supported model semantics. An assignment α is green, if G conforms to (C, sel) under α, and brown, if it does not. The shape catalogs are defined using the abstract syntax of the sim-

[4]At the time of writing, the repository contains pySHACL version 0.30.1, SHACL-S version 0.1.87, rudof version 0.3.18, Apache Jena 5.3.0, SHACL-TQ version 1.4.4 and ShEx-s version 0.2.34

ple shape languages. The conversion of those catalogs to ShEx and SHACL is presented in the github repository.

Basic separation tests. These four tests are designed to distinguish a validator that employs *LFP* semantics from one that employs *GFP*, using minimal examples.

- `bsep1`: $G = \{(a,p,a)\}$, $\mathscr{C} = \{s : \exists p.s\}$, $sel = \{(a,s)\}$
  $\alpha_{LFP} = \emptyset$, $\alpha_{GFP} = \{(a,s)\}$.
- `bsep2`: $G = \{(a,p,a),(b,p,b)\}$, $\mathscr{C} = \{s : \exists p.s\}$, $sel = \{(a,s),(b,s)\}$,
  $\alpha_{LFP} = \emptyset$, $\alpha_{GFP} = \{(a,s),(b,s)\}$,
  $\alpha_1 = \{(a,s)\}$, $\alpha_2 = \{(b,s)\}$.
- `bsep3`: $G = \{(a,p,c),(b,p,c)\}$, $\mathscr{C} = \{s : s' \wedge \exists p.\top,\ s' : s \wedge \exists p.\top\}$, $sel = \{(a,s),(b,s)\}$,
  $\alpha_{LFP} = \emptyset$, $\alpha_{GFP} = \{(a,s),(b,s),(a,s'),(b,s')\}$,
  $\alpha_1 = \{(a,s),(a,s')\}$, $\alpha_2 = \{(b,s),(b,s')\}$.
- `bsep4`: G as in `bsep2`, $\mathscr{C} = \{s : \exists p.s, s' : \neg s\}$, $sel = \{(a,s),(b,s)\}$,
  $\alpha_{LFP} = \{(a,s'),(b,s')\}$, $\alpha_{GFP} = \{(a,s),(b,s)\}$,
  $\alpha_1 = \{(a,s),(b,s')\}$, $\alpha_2 = \{(b,s),(a,s')\}$.

Reachability separation tests. The next four tests check reachability and its dual property of safety by checking them on two dual shape names r and s, in four different scenarios. All 4 tests use the same graph: $G = \{(a,p,d),(b,p,a),(b,p,c),(c,p,d),(d,p,c)\}$

- `reach1`: $\mathscr{C} = \{r : \text{test}(a) \vee \exists p.r\}$, $sel = \{(a,r),(b,r),(c,r),(d,r)\}$,
  $\alpha_{LFP} = \{(a,r),(b,r)\}$,
  $\alpha_{GFP} = \{(a,r),(b,r),(c,r),(d,r)\}$.
- `reach2`: $\mathscr{C} = \{s : \neg\text{test}(a) \wedge \forall p.s,\ r : \neg s\}$, $sel = \{(c,r),(d,r)\}$,
  $\alpha_{LFP} = \{(a,r),(b,r),(c,r),(d,r)\}$,
  $\alpha_{GFP} = \{(a,r),(b,r),(c,s),(d,s)\}$.
- `safe1`: $\mathscr{C} = \{s : \neg\text{test}(a) \wedge \forall p.s\}$, $sel = \{(c,s),(d,s)\}$,
  $\alpha_{LFP} = \emptyset$, $\alpha_{GFP} = \{(c,s)(d,s)\}$.
- `safe2`: $\mathscr{C} = \{r : \text{test}(a) \vee \exists p.r,\ s : \neg r\}$, $sel = \{(c,r),(d,r)\}$,
  $\alpha_{LFP} = \{(a,r),(b,r),(c,s),(d,s)\}$,
  $\alpha_{GFP} = \{(a,r),(b,r),(c,r),(d,r)\}$.

Feature tests These target more specific properties of the validator, as we will explain after introducing them.

- `nstrat1` (non-stratified negation without cyclic data):
  $G = \{(a,p,b),(b,p,c),(c,p,d),(d,p,e)\}$, $\mathscr{C} = \{s : \exists p.\neg s\}$, $sel = \{(a,s),(b,s),(c,s),(d,s),(e,s)\}$
  Expected result: A validation engine can reject the catalog for non-stratified shapes.
- `nstrat2` (non-stratified negation with cyclic data):
  $G = \{(a,p,b),(b,p,a)\}$, $\mathscr{C} = \{s : \exists p.\neg s\}$, $sel = \{(a,s),(b,s)\}$
  Expected result: A validation engine can reject the catalog for non-stratified shapes.
- `cons1` (consistency): $G = \{(a,p,a)\}$, $C = \{s : \exists p.s', s' : \neg s\}$, $sel = \{(a,s),(a,s')\}$
  Expected result: The engine should reject because a can't conform to s and s' at the same time.
- `cons2` (consistency): G as in `nstrat2`, C as in `cons1`, $sel = \{(a,s),(a,s'),(b,s),(b,s')\}$
  Expected result: The engine should reject because a and b can't conform to s and s' at the same time.
- `fresh` (fresh constant support):
  $G = \{(a,p,b),(b,p,c),(c,p,a)\}$, $\mathscr{C} = \{s : \top\}$,
  $sel = \{(d,s)\}$
  Expected result: The engine can accept because any value conforms to shape s, although as d is not part of the graph, it could also reject it.

The `nstrat1` and `nstrat2` tests contain a combination of negation and recursion which is non-stratified. The tests are included in the test suite to check how engines behave on such inputs although the proposed semantics doesn't support non-stratified catalogs. `cons1` and `cons2` check for *logical consistency*. We have a catalog with two shapes, s and s', where s' is defined as the negation of s. Then we ask in the shape map that both s and s' be assigned to the same node. Since no such assignment is possible while being consistent with the catalog's semantics, a consistent validator should reject this possibility. Finally, the test `fresh` has a shape catalog with a shape assignment that is trivially satisfied, using the shape $\top$. In the shape map, we require that this shape is satisfied by a "fresh node", that is, a node that is not featured in the graph. This reflects something that both SHACL and ShEx permit: selecting nodes outside the input graph.

Duality tests These tests check the duality proposition presented in [3] between LFP and GFP. `dual1` uses a catalog which contains a negation, a conjunction and a universal quantifier, while `dual2` contains the dual shape that swaps negation by a direct test, conjunction by disjunction and $\forall$ by $\exists$. The resulting shape assignments for LFP and GFP are complementary.

- `dual1` (Combines existential and basic test):
  $G = \{(a, p, b), (b, p, a), (b, q, c), (d, p, d)\}$, $\mathscr{C} = \{s : \neg\text{test}(b) \wedge \forall p.s\}$, $sel = \{(d, s)\}$,
  $\alpha_{LFP} = \{(c, s)\}$,
  $\alpha_{GFP} = \{(c, s), (d, s)\}$.
- `dual2` (dual shape of `dual1` which contains a universal quantifier and basic test):
  $G =$ as in `dual1`, $\mathscr{C} = \{s : \text{test}(b) \vee \exists p.s\}$, $sel = \{(d, s)\}$,
  $\alpha_{LFP} = \{(a, s), (b, s)\}$,
  $\alpha_{GFP} = \{(a, s), (b, s), (d, s)\}$.

Non-recursive tests We added two control tests `norec1` and `norec2` to check that non-recursive shapes behave as expected. In both cases, the shape assignments for LFP, GFP, bSMS and cSMS are equivalent.

- `norec1` (basic non-recursive test):
  $G = \{(a, p, a)\}$, $\mathscr{C} = \{s : \exists p.s'\ s' : test(a)\}$, $sel = \{(a, s)\}$,
  $\alpha_{LFP} = \alpha_{GFP} = \{(a, s), (a, s')\}$
- `norec2` (non-recursive test expecting non-conformance):
  $G = \{(a, p, a), (b, p, b)\}$, $\mathscr{C} = \{s : \exists p.\neg s'\ s' : test(a)\}$, $sel = \{(a, s)\}$,
  $\alpha_{LFP} = \alpha_{GFP} = \{(b, s), (a, s')\}$

Table 1 represents the results of running sheval using example 2 and the recursive shapes test suite.

6. Discussion

Using Table 1, we can see which validators are consistent with which semantics. In the case of the non-recursive control tests `norec1` and `norec2` all validators behave as expected.

For recursive shapes and non-stratified shape catalogs, we see that the three ShEx validators, rudof, Jena ShEx and ShEx-S, are consistent with GFP, as prescribed by the official semantics [4].

The situation is less uniform in the SHACL ecosystem. pySHACL detects cycles in most of the cases except in `reach2` but it continues trying to validate and in most of the cases it returns a conforms=true except in `example 2` and `bsep3` in which cases it gives an error. SHACL-s fails for example 2 and bsep3 and has a timeout error. The semantics seems to follow a GFP. Jena SHACL detects recursive cycles, and continues attempting the validation returning a conforming validation report following the GFPsemantics. The behaviour of SHACL-TQ is a bit different, as it is not consistent with either GFP, LFP, bSMS, or cSMS, as can be seen. SHACL-TQ thus seems to follow a unique semantics which would require a more in depth look to the algorithm implemented and the source code. Regarding rudof

Test	rudof	ShEx-S	Jena	pySHACL	SHACL-S	Jena	SHACL TQ	rudof no cycles	rudof LFP	rudof GFP	GFP	LFP	bSMS	cSMS
example 2	●	●	●	C!	T	C+	■	C	■	●	●	■	●	■
bsep1	●	●	●	C+	●	C+	●	C	■	●	●	■	●	■
bsep2	●	●	●	C+	●	C+	●	C	■	●	●	■	●	■
bsep3	●	●	●	C!	T	C+	■	C	■	●	●	■	●	■
bsep4	●	●	●	C+	●	C+	●	C	■	●	●	■	●	■
reach1	●	●	●	C+	●	C+	■	C	■	●	●	■	●	■
reach2	■	■	■	■	■	C-	■	C	●	■	■	●	●	■
safe1	●	●	●	C+	●	C+	●	C	■	●	●	■	●	■
safe2	●	●	●	C+	●	C+	■	C	■	●	●	■	●	■
nstrat1	N	N	■	■	■	C-	■	N	N	N	–	–	–	–
nstrat2	N	N	●	■	■	C-	■	N	N	N	–	–	–	–
cons1	N	N	■	■	■	C-	■	N	N	N	–	–	–	–
cons2	N	N	●	■	■	C-	■	N	N	N	–	–	–	–
fresh	●	●	●	●	●	●	●	●	●	●	●	●	●	●
dual1	●	●	●	●	●	C+	●	C	●	●	●	■	●	■
dual2	●	●	●	C+	●	C+	●	C	●	●	●	■	●	■
norec1	●	●	●	●	●	●	●	●	●	●	●	●	●	●
norec2	■	■	■	■	■	■	■	■	■	■	■	■	■	■

Table 1
Results of ShEx and SHACL validators for recursive shapes catalogues:
● = all nodes conform to the expected shapes (conforms = true)
■ = some nodes don't conform to their expected shapes (conforms = false)
T = timeout error during validation
C = recursive cycles detected and validation stopped
C+ = recursive cycles detected, but validator continued and returned (conforms=true)
C- = recursive cycles detected, but validator continued and found errors (conforms=false)
C! = engine detected recursive cycles, attempted the validation and crashed
N = engine detected non-stratified recursive shapes and stopped
– = no expected semantics,

SHACL, it has a flag to indicate whether it stops with recursive shapes (rudof no cycles) and another one to indicate which semantics to use, GFP(marked as brave) or LFP(marked as cautious), and the behaviour of rudof SHACL is consistent with the semantics indicated by the flag.

In the case of non-stratified catalogs, ShEx-S and rudof (ShEx and SHACL) correctly reject such inputs. Jena ShEx does not reject them and attempts to validate, answering that the graph validates in some cases or rejects in others. `cons1` asks for a node to be assigned two shapes, where by definition one negates the other. A pass in this context means reporting a validation error, as no logically consistent shape assignment can satisfy the desired target assignment. `cons2` is a similar test, but involves two nodes instead of just one. Maybe this asymmetric behaviour of Jena ShEx for two very similar tests can be attributed to a software bug. For non-stratified catalogs, except rudof, the rest of the SHACL validators attempt the validation and return non-conformance.

In the case of the duality tests, the ShEx validators are consistent with GFP and the SHACL validators also seem to follow GFP.

Our test results do not prove that any of the validators follow a given semantics. They just prove that some implementations do not follow a given semantics, and they give us hints as to how these implementations behave. Claiming that their semantics follows GFP, LFP, bSMS, or cSMS requires knowledge of the algorithm each system implements, and a formal study of said algorithm.

7. Related work

This paper can be considered a sequel of [3] with a focus on a more practical description of the framework used in that paper for the experiments. That paper was preceded by a previous paper focused on finding a common foundation of schema languages like ShEx, SHACL and PG-Schema without recursion[11]. Combining recursive shapes and negation have been a significant topic of recent research. This is especially pronounced for SHACL, where the W3C recommendation [12] left the semantics of recursion undefined. The challenges of recursion were first formally addressed by Corman et al. [13, 14], who proposed a *supported model semantics* based on first-order logic. Subsequent work has explored various semantics inspired by paradigms from fixed-point logics and logic programming [15, 16, 17, 18, 19]. In contrast to SHACL, the semantics of recursion in ShEx has been consistently defined via *greatest fixed-points* [5, 2]. A previous minimal language common to ShEx and SHACL was the S language defined in [20] which was an inspiration for the Simple Shape Language.

There have been several efforts to define benchmarks and test suites for shape languages. One of the earliest was the Webindex benchmark proposed in [21] which included both a ShEx and SHACL generator benchmark tool with recursive shapes. Those shapes were inspired by a real-world use case and didn't explore the different semantics associated.

SHACL recommendation refers to the SHACL test suite [22] which contains a set of test cases for SHACL engines. The test suite doesn't include tests for recursion, and in case future work on SHACL tackled the need for recursive validation, it could be extended with the recursive test suite presented in this paper. The ShEx test suite [23] contains more than one thousand test cases for ShEx engines, including some recursive shapes, but it doesn't explore different semantics of recursion apart of GFP.

8. Conclusions

We described SHEVAL, a tool to run test suites for the evaluation of different shape engines. It has been used to evaluate and understand the differences in the implementation of recursive shapes in ShEx and SHACL. SHEVAL provides a framework for testing and comparing the behaviour of different shape engines against a set of expectations, helping to identify inconsistencies and potential issues, and providing a basis for further research and development in the field of shape-based validation of RDF data. The tool is easily customizable and generates a detailed report which can be serialized in YAML, CSV or LaTeX so it can be later analysed.

Future work includes the addition of other ShEx and SHACL implementations. Another line for future work is to add and define additional test suites and cover other aspects of shape-based validation, such as expressiveness, interoperability, performance, explainability or usability.

Acknowledgments

This work has been partially funded by the project SHAKIGRA: Shaping Knowledge and Interoperable Graphs, code: NAC-ES-PUB-ASV-2025 PID2024-157010OBI00, from the Spanish Research Agency, the regional project with code SEK25-GRU-GIC-24-089 and COST Action CA23147 GOBLIN - Global Network on Large-Scale, Cross-domain and Multilingual Open Knowledge Graphs (Jose E. Labra-Gayo). This was partially supported by the Province of Bolzano and FWF through project OnTeGra (DOI 10.55776/PIN8884924, Savković).

Declaration on Generative AI

During the preparation of this work, the author(s) used a combination of ChatGPT, Copilot and Claude Code in order to: Grammar and spelling check, Paraphrase and reword and Improve writing style. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] E. Prud'hommeaux, J. E. Labra Gayo, H. Solbrig, Shape expressions: an RDF validation and transformation language, in: International Conference on Semantic Systems (SEM), ACM, 2014, pp. 32–40. URL: https://doi.org/10.1145/2660517.2660523. doi:10.1145/2660517.2660523.

[2] S. Staworko, I. Boneva, J. E. Labra Gayo, S. Hym, E. Prud'hommeaux, H. Solbrig, Complexity and expressiveness of ShEx for RDF, in: International Conference on Database Theory (ICDT), 2015, pp. 195–211.

[3] S. Ahmetaj, I. Boneva, J. Hidders, M. Jakubowski, J.-E. Labra-Gayo, W. Martens, F. Mogavero, F. Murlak, C. Okulmus, O. Savković, M. Šimkus, D. Tomaszuk, Common foundations for recursive shape languages, in: 23rd International Conference on Principles of Knowledge Representation and Reasoning, KR'26, 2026. URL: https://arxiv.org/abs/2604.20946. arXiv:2604.20946.

[4] E. Prud'hommeaux, I. Boneva, J. E. Labra Gayo, G. Kellog, Shape Expressions Language 2.1, W3C Community Group Report, W3C, 2019. Http://shex.io/shex-semantics/.

[5] I. Boneva, J. E. Labra Gayo, E. G. Prud'hommeaux, Semantics and validation of shapes schemas for RDF, in: ISWC, 2017, pp. 104–120.

[6] TopQuadrant SHACL, TopQuadrant SHACL, https://github.com/TopQuadrant/shacl, 2025. Accessed: 2025-10-04.

[7] Apache Jena, Apache Jena, https://jena.apache.org/, 2025. Accessed: 2025-10-04.

[8] RuDoF Project, Rudof project, https://github.com/rudof-project/rudof, 2025. Accessed: 2025-10-04.

[9] pySHACL, pyshacl: Python validator for SHACL, https://github.com/RDFLib/pySHACL, 2025. Accessed: 2025-10-04.

[10] SHACL-S, SHACL-s, https://github.com/weso/shacl-s, 2024.

[11] S. Ahmetaj, I. Boneva, J. Hidders, K. Hose, M. Jakubowski, J. E. L. Gayo, W. Martens, F. Mogavero, F. Murlak, C. Okulmus, A. Polleres, O. Savkovic, M. Simkus, D. Tomaszuk, Common foundations for SHACL, ShEx, and PG-Schema, in: The Web Conference (WWW), 2025, pp. 8–21.

[12] H. Knublauch, D. Kontokostas, Shapes Constraint Language (SHACL)., W3C Recommendation, W3C, 2017. https://www.w3.org/TR/shacl/.

[13] J. Corman, J. L. Reutter, O. Savković, Semantics and validation of recursive SHACL, in: ISWC, 2018, pp. 318–336.

[14] J. Corman, F. Florenzano, J. L. Reutter, O. Savkovic, Validating SHACL constraints over a SPARQL endpoint, in: ISWC, 2019, pp. 145–163.

[15] M. Andresel, J. Corman, M. Ortiz, J. L. Reutter, O. Savkovic, M. Šimkus, Stable model semantics for recursive SHACL, in: The Web Conference (WWW), 2020, pp. 1570–1580.

[16] C. Okulmus, M. Šimkus, SHACL validation under the well-founded semantics, in: KR, 2024, pp. 553–562.

[17] B. Bogaerts, M. Jakubowski, Fixpoint semantics for recursive SHACL, in: ICLP, 2021, pp. 41–47.

[18] S. Ahmetaj, B. Löhnert, M. Ortiz, M. Simkus, Magic shapes for SHACL validation, Proc. VLDB Endow. 15 (2022) 2284–2296.

[19] P. Pareti, G. Konstantinidis, F. Mogavero, Satisfiability and Containment of Recursive SHACL, JWS 74 (2022) 100721:1–24.

[20] J. E. Labra Gayo, H. García-González, D. Fernández-Alvarez, E. Prud'hommeaux, Challenges in RDF validation, in: Current Trends in Semantic Web Technologies: Theory and Practice, 2019, pp. 121–151.

[21] J. E. Labra Gayo, E. Prud'hommeaux, H. Solbrig, I. Boneva, Validating and describing linked data portals using shapes, CoRR abs/1701.08924 (2017). URL: https://arxiv.org/abs/1701.08924.

[22] J. E. Labra Gayo, H. Knublauch, D. Kontokostas, SHACL Test Suite and Implementation Report, W3C Document, W3C, 2017. https://w3c.github.io/data-shapes/data-shapes-test-suite/.

[23] ShEx Community Group, shextest: ShEx test suite, http://shexspec.github.io/shexTest/, 2017. Accessed 2026-07-19.