SoftwareX 34 (2026) 102592

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

sheXer: A robust and versatile library for automatic extraction of RDF shapes

Daniel Fernández-Álvarez*, Jose Emilio Labra-Gayo

University of Oviedo, WESO Research Group, Oviedo, Spain

ARTICLE INFO

Keywords:
RDF
RDF shapes
ShEx
SHACL

ABSTRACT

RDF shapes are effective tools for describing and validating RDF content. However, producing and maintaining them is a time-consuming task that typically requires domain experts with technical skills.

In this paper, we present sheXer, an open-source Python library for the automatic extraction of RDF shapes from existing RDF instance data. sheXer supports a wide range of use cases by accepting multiple input formats (e.g., text files, SPARQL endpoints, compressed content), producing different types of outputs (SHACL, ShEx, UML visualizations, RDF-Config files, statistics), and enabling extractions tailored to specific goals.

Metadata

The link provided as the Reproducible Capsule in Table 1 points to a folder containing several Jupyter notebooks with runnable code.

Table 1
Code metadata (mandatory).

Nr.	Code metadata description	Metadata
C1	Current code version	2.6.5
C2	Permanent link to code/repository used for this code version	https://github.com/weso/shexer/archive/refs/tags/2.6.5.zip
C3	Permanent link to Reproducible Capsule	https://github.com/weso/shexer/ tree/b04627c1fc7cf7867bca1c14600e5 911cc3ecc4a/doc
C4	Legal Code License	Apache 2.0 License
C5	Code versioning system used	Git
C6	Software code languages, tools, and services used	Python
C7	Compilation requirements, operating environments & dependencies	sheXer can be installed with "pip" in any Operating System compatible with Python 3.7 or newer versions. Dependencies to other Python libraries: rdflib, SPARQLWrapper, wLighter, plantuml, python-xz, six, Flask, Flask-Cors.
C8	If available Link to developer documentation/manual	Documentation at: https://github.com/weso/shexer/blob/master/README.md. Executable example code at: https://github.com/weso/shexer/tree/master/doc
C9	Support email for questions	fernandezalvdaniel@uniovi.es

1. Motivation and significance

RDF shape languages describe the expected structure of an RDF source. This structure is defined in terms of the types and amounts of relations that should or may occur between nodes in an RDF graph. Shapes Constraint Language (SHACL) [25] and Shape Expressions (ShEx) [32] are two such languages that have proven effective for validating and documenting RDF data [29]. Despite these benefits, the adoption of RDF shapes remains limited compared to RDF itself. This is due to several challenges, including:

- RDF shapes require continuous maintenance as data schemas evolve.
- Writing shapes demands technical expertise in RDF technologies and knowledge of the target domain.

* Corresponding author.
Email addresses: fernandezalvdaniel@uniovi.es (D. Fernández-Álvarez), labra@uniovi.es (J.E. Labra-Gayo).

https://doi.org/10.1016/j.softx.2026.102592
Received 1 September 2025; Received in revised form 14 February 2026; Accepted 1 March 2026
Available online 12 March 2026
2352-7110/© 2026 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

```
ex:PlayerShape {
  rdf:type  [ ex:Player ] ; # It must have type ex:Player
  schema:name  xsd:string ;  # string name
  schema:birthDate  xsd:date?  ; # optional birthdate
  ex:team  @ex:TeamShape ; # reference to ex:TeamShape
  ex:injury  @ex:InjuryShape * ; # zero or more injuries
}

```

Fig. 1. Potential shape for the class *ex:Player* expressed in ShEx.

- The effort needed to create and maintain schemas grows with the size and heterogeneity of the data, making producing shapes for large graphs particularly challenging.

To address these issues, automatic shape extractors have emerged. These tools generate shapes automatically by exploiting existing knowledge, such as the RDF graph itself or related ontologies. In this paper, we present *sheXer* [18], an open-source Python library for the automatic generation of RDF shapes. It has been under active development since 2018 [15,17] and has evolved into a robust and versatile tool capable of processing large volumes of data on commodity hardware.

sheXer can extract data from multiple sources, including local and remote RDF files, SPARQL endpoints, RDFLib graphs, and their combinations. It supports the generation of both ShEx and SHACL shapes, as well as UML-based schema visualizations and RDF-Config files, and offers extensive configuration options to adapt the extraction process. The library can be installed using standard Python tools (e.g., *pip*).

sheXer's repository includes documentation and example code.[1] Compared to existing alternatives, sheXer provides a distinctive and actively evolving feature set, driven by community requirements and designed to scale to large datasets.

2. Software description

One of the main uses of shape languages, such as ShEx and SHACL, is RDF validation [29]. The usual workflow is that a domain expert writes shapes, and those shapes are then used to validate some content. In contrast, sheXer analyzes an existing RDF graph in order to discover patterns (shapes) in the data. It assumes that certain subsets of nodes in the analyzed graph share a common pattern, and then mines those nodes and their connections to find the most precise shape that conforms to them all. The pattern identified for each subset of nodes is expressed as a shape.

To achieve this, sheXer requires the user to provide at least: 1) an input graph, and 2) *indications* on how to identify appropriate node subsets from which to extract a shape.[2] These indications are specified using the different configuration parameters of sheXer.

Consider, for example, an RDF graph containing information about a sports league. A user may want to discover which properties and node types are connected to nodes representing professional players in such a graph. sheXer can be used to obtain this knowledge by providing the graph itself as input, along with indications to extract a shape that conforms to every node that is an instance of the class **ex:Player**.

A possible shape describing the instances of **ex:Player** is shown in Fig. 1. This shape can be used for different purposes, including identifying the appropriate properties and structures needed to build suitable SPARQL queries.

2.1. Core algorithm

sheXer aims to extract shapes that conform, as precisely as possible using *open* shapes [25,32], to the neighborhoods of all elements belonging to a given subset of nodes in an RDF graph.

Note that the shapes produced by this process are expected to be valid only within the context of the analyzed graph. If we extract a shape associated with the instances of the class **ex:Person** for a given graph $g1$, the constraints of that shape will be based solely on the information observed in $g1$. If we have another graph $g2$ that also uses the class **ex:Person**, but whose instances of **ex:Person** exhibit different neighborhood topologies compared to those in $g1$, then the shape associated with **ex:Person** in $g2$ must be different from the one derived from $g1$.

To explain sheXer's core algorithm, we will use the following nomenclature:

- We define a Graph g as a tuple $g = (N, E)$, where N is a set of nodes and E is a set of edges. Each edge e is a triple $e = (n_1, p, n_2) : n_1 \in N \wedge n_2 \in N$, and $e \in (IRI \cup BlankNode) \times IRI \times (IRI \cup BlankNode \cup Literal)$. A triple expresses a directed connection between n_1 and n_2 using a property p. The elements in a triple are referred as *subject*, *predicate*, and *object*.
- We define a target declaration N_t over a graph $g = (N, E)$ as a set of nodes $N_t \mid N_t \subset N$.
- We define the set of relevant nodes for a sheXer execution N_T as the union of the nodes in a set T of target declarations over a graph g. Formally, $N_T = \bigcup_{t \in T} N_t$
- We define the neighborhood of a node $n \in N$ in the context of a graph $g = (N, E)$ as the set of edges E_n where n is subject or object, i.e., $\forall e \in E_n,\ e = (n_1, p, n_2) \leftrightarrow (n_1 = n \ \vee \ n_2 = n)$.
- We define a Shape s associated with a graph $g = (N, E)$ as a tuple (l, C_s, r), where l is a label, C_s is a set of constraints, and r can be either an empty element or a textual regular expression.[3]
- A constraint $c \in C_s$ is tuple (p, t, u), where:
  - p is a property used as predicate in any edge of E.
  - t is a type. In this context, a type can refer to a shape label, a literal type, a macro defined in ShEx/SHACL or a logical expression involving several individual types.
  - u is a cardinality, which is a pair of natural numbers (a, b) expressing a range ($b \geq a$). b could be the special value *unbounded*.

sheXer follows a four-step strategy:

1. Traverse the source to identify the nodes belonging to each target declaration N_t. How this is done depends on the nature of the instructions provided by the user to define target declarations. For example, when sheXer is asked to extract a shape for each class, populating these sets consists of locating the instances of

[1] https://github.com/weso/shexer Accessed on March 4, 2026.

[2] The extraction process can be configured with many other parameters, which are described in Section 2.3.

[3] This definition does not describe the full expressivity of a ShEx/SHACL shape. It is a simplification that is useful for describing sheXer's algorithms, adapted to sheXer's potential outputs.

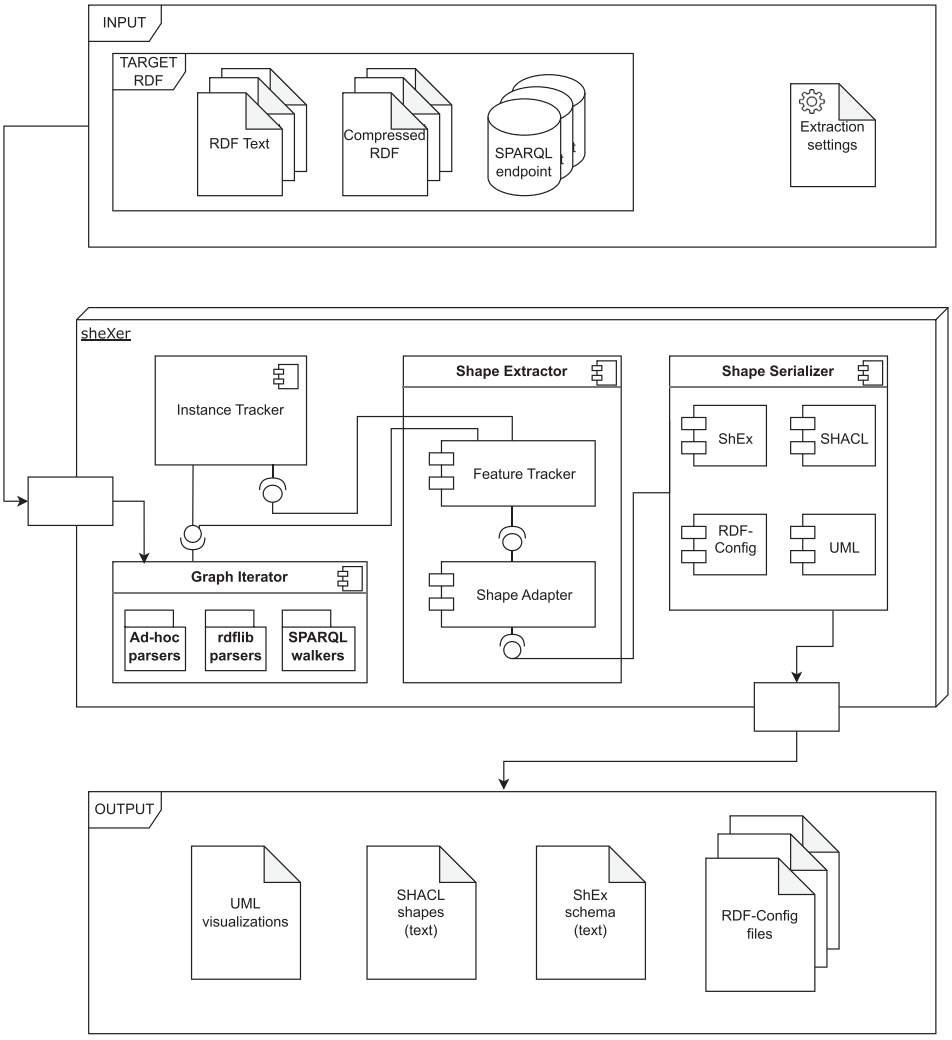

Fig. 2. sheXer's architecture.

such classes. Identifying the members of each target declaration makes it possible to build the set N_T.

2. Traverse the input graph to store the features observed in the neighborhood of each node $n_i \in N_T$. By features, we mean the predicates used, the types of connected nodes (literals, BNodes, nodes conforming to shapes, etc.), and the observed frequencies of each property-type combination. This information can be found in those triples in which any $n_i \in N_T$ appears as subject or object.
3. For each target declaration N_{t_i}, analyze the features of every node in N_{t_i} and derive a set of constraints C_{s_i} that all nodes conform to, aiming for maximal precision w.r.t. property usage, cardinality, and type specificity. Then, create a shape s_i by associating C_{s_i} with a descriptive label and, only if requested by the user, a regex pattern matching all node IRIs in N_{t_i}.
4. Serialize each shape s_i, together with its associated constraints C_{s_i}, label, and regex pattern, in the user-selected format (ShEx, SHACL, UML, etc.).

This approach applies to both RDF data and SPARQL endpoints and does not require loading the entire input into memory. Steps 1 and 2 traverse the source independently. When mining SPARQL endpoints, subsets of the exposed content are retrieved for local traversal. Although this may introduce overhead when extracting a small number of shapes from very large sources, it significantly reduces memory usage and enables the processing of large graphs on commodity hardware.

Two traversals are required to support linking constraints in which the type t is the identifier (label) of another shape, since the members of every target declaration N_{t_i} must be known before Step 2. This may introduce issues with blank nodes, as they lack universal and consistent identifiers. However, for most input types—except for some Turtle serializations—sheXer employs internal strategies to assign ad hoc, semantically meaningless identifiers (e.g., sequential integers), allowing such nodes to be consistently referenced across independent traversals.

2.2. Architecture and implementation

sheXer adopts a modular architecture composed of four components: the *Graph iterator*, which traverses the source in Steps 1 and 2; the *Instance tracker*, which finds the members of each target declaration N_{t_i} — and therefore the set N_T which is the union of them all; the *Shape extractor*, which associates a shape to each target declaration in Step 3; and the *Shape serializer*, which produces the outputs in Step 4. Fig. 2 illustrates this architecture.

The *Graph iterator* is responsible for traversing the input source and the library includes several implementations to support different scenarios. Parsing can be performed using RDFLib [26] or via ad-hoc parsers for Turtle and N-Triples that avoid loading the entire input into memory. Additional iterators support SPARQL endpoints, compressed files, and combinations of all these inputs so they can be processed as a single graph.

The *Instance tracker* identifies nodes belonging to each target declaration. This is done by producing a mapping from node URIs to the shape label(s) associated with the target declaration(s) in which the node is included. For that, the graph is traversed with the *Graph iterator*.

The *Shape extractor* performs Step 3 using a voting mechanism in which features observed in node neighborhoods vote for candidate constraints with varying precision. For example, if a node is connected via property p to two nodes conforming to shape s_1, it may vote for constraints with elements such as: 1) type s_1 with cardinality *exactly 2*; 2) type *IRI* with cardinality *exactly 2*; or 3) type s_1 with cardinality *one or more*. After voting, the most precise constraints supported by all nodes of a target declaration are chosen. If no constraint is universally supported, the most precise and widely supported alternative is chosen and its cardinality is relaxed to allow zero occurrences. Voting is handled by the *Feature extractor* sub-module, while consolidation of votes is performed by the *Shape adapter*.

The *Shape extractor* uses the mapping produced by the *Instance tracker* to determine which nodes contribute to each shape and explores their neighborhoods using the *Graph iterator*. Once an internal abstract schema is produced, the *Shape serializer* generates the final outputs in the requested format(s).

Performance evaluations across various data volumes and additional details on sheXer's core algorithms are provided in [17].

2.3. Features

- **Regarding input**:
  - Different types of inputs (files, endpoints, compressed content, RDFLib graphs, etc.).
  - Support for all main RDF syntaxes.
  - Support for iterative parsing to address scalability issues.
  - Support for data sampling.
- **Outputs**:
  - Both ShEx and SHACL.
  - UML visualizations.
  - RDF-Config files.
  - Special shapes describing federated input sources.
  - Statistics provided in comments (currently, only in ShEx).
  - Annotation of example nodes or IRI patterns (regex) conforming to shape/constraints.
- **Regarding the extraction process**:
  - Different configuration parameters to declare how to build target declarations (including white list of classes, all classes with at least an instance, shape maps [31] for both ShEx and SHACL, or SPARQL selectors).
  - Definition of an acceptance threshold to filter rarely observed features.
  - Extraction of inverse paths (when the node to validate acts as the object in a triple).
  - Configurable precision in cardinalities.
  - Different strategies to handle constraints that are not universally met among the example nodes.
  - Different settings regarding endpoint exploration:
    - Depth from seed (target) nodes.
    - Depth exception for typing triples.
    - Use of a local cache to avoid redundant queries.
  - Selection of a property to express a class-instance relation than **rdf:type**.
  - Blacklist to ignore triples with properties in certain namespaces.
  - Specific adaptations for the Wikidata data model, such as annotation of identifiers with human-readable labels or adapted generation of shapes for qualifiers.[4]

To further improve scalability, we developed a slice–consolidate approach [20] that enables processing large sources without sampling. Using this method, we extracted shapes from a substantial portion of UniProt, estimated at 15.9 billion triples [19]. The scripts implementing such approach are publicly available.[5]

3. Illustrative examples

3.1. How to use sheXer

Running sheXer requires instantiating a single Python object, `Shaper`, and invoking its `shex_graph` method. This basic workflow can be configured using up to 57 parameters, which can be provided either in the constructor or in the `shex_graph` call.

Fig. 3 shows example code. In Example 1, shapes are extracted from a file located at `local_graph`. `all_classes_mode=True` causes to generate one shape for each class with at least one instance. Setting `string_output=True` causes `shex_graph` to return the result as a Python `str`. `namespaces_dict` defines prefix–namespace mappings to be used in the output.

Assuming the input file at `local_graph` contains the RDF shown in Fig. 4, the output of Example 1 is presented in Fig. 5.

Example 2 of Fig. 3 expands Example 1 with additional parameters:

- `examples_mode=ALL_EXAMPLES`: Instructions on how to build target declarations. This setting tells sheXer to build a target declaration for each class with at least one declared instance. Each target declaration will contain instances of a single class.
- `instances_cap=3`: limit the size of each target declaration to three nodes at most (sampling).
- `instances_report_mode=MIXED_INSTANCES`: report conformance statistics as both ratios and absolute counts.
- `detect_minimal_iri=True`: infer an IRI regex pattern valid for all nodes of each target declaration.
- `acceptance_threshold=0.5`: exclude from the resulting shapes features observed in fewer than 50% of the members of a target declaration.
- `generate_annotations=True`: provide machine-readable RDF annotations instead of inline comments for any content beyond mere constraints.

The result of Example 2 is shown in Fig. 6.

sheXer's GitHub repository provides several Jupyter notebooks illustrating how to configure all features listed in Section 2.3.[6]

3.2. Real use cases

Extracted RDF shapes can be used to document RDF graphs, enabling users to add new data consistently and to write queries aligned with existing structures. Representative use cases are described in [7,38].

Shapes generated with sheXer have also been applied to validation and data quality assessment, for example, in the life sciences [1,41] and cultural heritage [3–5]. For instance, in [1], sheXer was used to extract ShEx schemas from RDF generated from heterogeneous non-RDF sources, enabling structural validation of the data produced, as well as detection of data outliers through examination of statistical annotations.

sheXer can also be used to produce artifacts with purposes different from describing or validating RDF content, such as RDF-Config [23]. RDF-Config specifications can be used to automatically produce SPARQL queries, diagrams, ShEx schemas, and configuration files for other tools. Thus, the combination of sheXer with RDF-Config enables pipelines that derive these artifacts directly from RDF data.

[4] each statement in Wikidata can be qualified with metadata. Translating such model into RDF requires reification [22] and produces more complex structures.

[5] https://github.com/shex-consolidator/shex-consolidator Accessed on March 4, 2026.

[6] https://github.com/weso/shexer/tree/master/doc Accessed on March 4, 2026.

```
from shexer.shaper import Shaper
local_graph = "/disk/path/to/local/ntriples_file.nt"
namespaces = {"http://example.org/": "ex",
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#": "rdf",
    "http://www.w3.org/2001/XMLSchema#": "xsd",
    "http://xmlns.com/foaf/0.1/": "foaf",
    "http://weso.es/shexer/ontology/": "shexer",
    "http://www.w3.org/2000/01/rdf-schema#" : "rdfs"
}
####### EXAMPLE 1 - BASIC #######
shaper = Shaper(all_classes_mode=True,
                namespaces_dict=namespaces,
                graph_file_input=local_graph)
result = shaper.shex_graph(string_output=True)
print(result)

####### EXAMPLE 2 - SOME EXTRA PARAMETERS #######
from shexer.consts import TURTLE, ALL_EXAMPLES,
    MIXED_INSTANCES
shaper = Shaper(
            graph_file_input=local_graph,
            namespaces_dict=namespaces,
            examples_mode=ALL_EXAMPLES,
            instances_cap=3,
            instances_report_mode=MIXED_INSTANCES,
            generate_annotations=True,
            all_classes_mode=True,
            detect_minimal_iri=True,
            input_format=TURTLE)
str_result = shaper.shex_graph(string_output=True,
                               acceptance_threshold=0.5)
print(str_result)
```

Fig. 3. Examples of how to run sheXer.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:Jimmy a foaf:Person ;
  foaf:age 23 ;
  foaf:name "Jimmy" ;
  foaf:familyName "Jones" .

ex:Sarah a foaf:Person ;
  foaf:age 22 ;
  foaf:name "Sarah" ;
  foaf:familyName "Salem" .

ex:Bella a foaf:Person ;
  foaf:age 56 ;
  foaf:name "Isabella" .

ex:David a foaf:Person ;
  foaf:name "David" ;
  foaf:familyName "Doulofeau" ;
  foaf:knows ex:Sarah .
```

Fig. 4. Input example.

```
PREFIX ex: <http://example.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX : <http://weso.es/shapes/>

:Person
{
   rdf:type  [foaf:Person]  ;
     # 100.0
   foaf:name  xsd:string  ;
     # 100.0
   foaf:age  xsd:integer  ?;
             # 75.0
   foaf:familyName  xsd:string  ?;
             # 75.0
   foaf:knows  @:Person  ?
             # 25.0
}
```

Fig. 5. Output of example 1 in Fig. 3.

```
:Person  [<http://example.org/>~]  AND
{
   rdf:type  [foaf:Person]    //  shexer:ratio_property_usage  1.000
    //  shexer:total_conforming_instances  3  ;
   foaf:age  xsd:integer    //  shexer:ratio_property_usage  1.000
    //  shexer:total_conforming_instances  3
    //  shexer:conformant_example  23  ;
   foaf:name  xsd:string    //  shexer:ratio_property_usage  1.000
    //  shexer:total_conforming_instances  3
    //  shexer:conformant_example  "Jimmy"  ;
   foaf:familyName  xsd:string  ?  //  shexer:ratio_property_usage  0.667
    //  shexer:total_conforming_instances  2
    //  shexer:conformant_example  "Jones"
    //  rdfs:comment "66.666
} // shexer:conformant_example ex:Jimmy
  // shexer:total_conforming_instances 3
```

Fig. 6. Output of example 2 in Fig. 5.

Additional real and potential applications of shapes extracted with sheXer are described in [14,27,39].

4. Impact

4.1. Comparison with alternatives

Several different alternatives have been proposed for automatic shape extraction, including QSE [33], COSE [10], ShapeDesigner [2], SHACLearner [21], RDFShapeInduction [30], RDFMining [13], *Spahiu et al.* [36], SHACLGen [24], SCOOP [11], RML2SHACL [9], Astrea [8], XSD2SHACL [12], s2s [35], and *Carreiro et al.* [6].

Table 2 summarizes selected features of these systems. As can be observed, sheXer includes capabilities not found in existing alternatives. Although sheXer was among the first publicly available tools for shape extraction, it remains relevant due to its distinctive feature set and its proven ability to support real-world use cases.

4.2. Indicators of maintenance and adoption

Since 2016, we have published several works on automatic shape extraction [15–17,20] while actively maintaining sheXer. sheXer's repositories have accumulated 704 commits,[7] and 62 releases have been published on PyPI.[8]

Some indicators of sheXer's adoption include the following[9]:

- According to the PyPI statistics API,[10] sheXer was downloaded 2823 times in the previous month.
- sheXer's GitHub repository has received 43 stars,[11] the highest number among public shape extraction repositories known to us.
- GitHub reports 54 repositories and 4 packages with explicit dependencies on sheXer.[12]

[7] At the time of writing, sheXer's main repository (https://github.com/weso/shexer) contains 526 commits. Earlier versions were hosted in a separate repository (https://github.com/DaniFdezAlvarez/shexerp3), archived since 2021 but still publicly accessible, which received 179 commits. Accessed on March 4, 2026.

[8] https://pypi.org/project/shexer/#history Accessed on March 4, 2026.

[9] These figures may change over time. The PyPI statistics API does not provide historical snapshots, and GitHub does not offer permanent links for past stars or dependency metrics.

[10] https://pypistats.org/packages/shexer Accessed on March 4, 2026.

[11] https://github.com/weso/shexer/stargazers Accessed on March 4, 2026.

[12] https://github.com/weso/shexer/network/dependents Accessed on March 4, 2026. GitHub informs that these figures are approximate.

Table 2
Summary of features of different shape extraction tools.

		sheXer	QSE	SCOOP	COSE	ShapeDesigner	SHACLearner	RDFShapeInduction	Spahiu et al.	RDFMining	SHACLGen	RML2SHACL	Astrea	XSD2SHACL	s2s	Carreiro et al.
OUTPUTS	Generation of SHACL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
	Generation of ShEx	✓				✓		✓								
	Generation of stats	✓				✓			✓[a]							
	UML visualization	✓														
	Generation of RDF-Config	✓														
	Federated shapes[b]	✓														
	Generation of annotated examples	✓														
	Visualization with UI		✓[c]			✓	✓[d]									
	Non-standard patterns															✓
INPUTS	RDF files (A-BOX)	✓	✓		✓	✓	✓	✓	✓	✓	✓					
	Ontologies (T-BOX)			✓							✓		✓			
	SPARQL endpoints	✓									✓					
	Mappings (such as RML)			✓								✓				
	Data schemas (such as XSD)			✓										✓		
	SHACL shapes + CONSTRUCT queries														✓	
	Dynamic graphs[e]				✓											
	Wikidata graphs[f]															✓
	Mechanisms for handling noise	✓	✓			✓	✓			✓						✓
	Mechanisms for handling large graphs	✓	✓	✓	✓		✓						✓	✓	✓	
	Inverse paths	✓					✓									

[a] Obtained from ABSTAT [37].
[b] Shapes extracted for environments with federated queries.
[c] Achieved by integration with SHACTOR [34].
[d] Achieved by integration with Schimatos [40].
[e] Shape extraction over different versions of a graph by computing only the delta.
[f] It refers to Wikidata's non-RDF data model, including qualifiers. Other proposals can handle Wikidata's RDF representation.

5. Conclusions

In this document, we have presented sheXer, a tool for the automatic extraction of RDF shapes from existing RDF data. sheXer is an open-source Python library that can parse diverse and potentially large inputs and generate shapes in both SHACL and ShEx. It can also produce artifacts for data maintenance and documentation, as well as alternative schema representations, including statistical reports, UML visualizations, and RDF-Config files.

We reviewed sheXer's technical proposals, outlined indicators of its adoption, described its core functionality, and presented real-world use cases. Overall, sheXer emerges as a mature and versatile tool that has been successfully applied in a wide range of automatic shape extraction scenarios. We consider it a valuable resource for RDF documentation and maintenance tasks, which can be also used for the automated generation of artifacts from RDF data structures.

We are currently exploring several lines for future work, including integration with strategies based on ontology inference, improvement of the slice–consolidation strategy described in Section 2.3, and re-implementation in Rust to further improve scalability by leveraging this language's performance and memory efficiency. We are currently working on the integration of sheXer with the Rust library rudof [28].

CRediT authorship contribution statement

Daniel Fernández-Álvarez: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Jose Emilio Labra-Gayo:** Writing – review & editing, Project administration, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially funded by projects *ANGLIRU*: Applying kNowledge Graph store search data interoperabiLIty and ReUsability, code: PID2020 - 117912RB and *SHAKIGRA*: Shaping Knowledge and Interoperable Graphs, code: NAC-ES-PUB-ASV-2025 PID2024-157010OB-I00, both from the Spanish Research Agency, and the regional project: SEK-25-GRU-GIC-24-089.

References

[1] Arakawa K, Ono T, Aoki-Kinoshita KF, Yamamoto Y. Development of an integrated and inferenceable RDF database of glycan, pathogen and disease resources. Sci Data 2023;10(1):582.
[2] Boneva I, Dusart J, Alvarez DF, Gayo JEL. Shape designer for ShEx and SHACL constraints. In: ISWC 2019-18th international semantic web conference; 2019.
[3] Candela G. An automatic data quality approach to assess semantic data from cultural heritage institutions. J Assoc Inf Sci Technol 2023;74(7):866–78.
[4] Candela G. Towards a semantic approach in GLAM labs: the case of the data foundry at the national library of scotland. J Inf Sci 2023:01655515231174386.
[5] Candela G, Pereda J, Sáez D, Escobar P, Sánchez A, Torres AV, Palacios AA, McDonough K, Murrieta-Flores P. An ontological approach for unlocking the colonial archive. ACM J Comput Cult Herit 2023;16(4):1–18.
[6] Carriero VA, Groth P, Presutti V, et al. Towards improving wikidata reuse with emerging patterns. In: CEUR workshop proceedings, vol. 3262. CEUR Workshop Proceedings; 2022. p. 1–12.
[7] Cifuentes-Silva F, Fernández-Álvarez D, Labra-Gayo JE. National budget as linked open data: new tools for supporting the sustainability of public finances. Sustainability 2020;12(11):4551.
[8] Cimmino A, Fernández-Izquierdo A, García-Castro R. Astrea: automatic generation of SHACL shapes from ontologies. In: European semantic web conference. Springer; 2020. p. 497–513.
[9] Delva T, De Smedt B, Min Oo S, Van Assche D, Lieber S, Dimou A. RML2SHACL: RDF generation taking shape. In: Proceedings of the 11th knowledge capture conference; 2021. p. 153–60.
[10] Dhouib B, Appuswamy R. COSE: continuous shapes extraction from dynamic knowledge graphs. In: IEEE, editor. BIGDATA 2025, IEEE international conference on big data, 8–11 December 2025, Macau SAR, China. Macau; 2025.
[11] Duan X, Chaves-Fraga D, Derom O, Dimou A. SCOOP all the constraints' flavours for your knowledge graph. In: Proceedings of the 21th extended semantic web conference (ESWC); 2024.
[12] Duan X, Chaves-Fraga D, Dimou A. XSD2SHACL: capturing RDF constraints from XML schema. In: Proceedings of the 12th knowledge capture conference 2023; 2023. p. 214–22.
[13] Felin R, Monnin P, Faron C, Tettamanzi AGB. An algorithm based on grammatical evolution for discovering SHACL constraints. In: European conference on genetic programming (part of EvoStar). Springer; 2024. p. 176–91.
[14] Fernández-Álvarez D, Bolleman J, Labra-Gayo JE, Yamamoto Y, Waagmeester A, Nishida K, Nagazumi H, Queralt-Rosinach N, Micklem G, Wu C. BioHack24 report: using discovered RDF schemes: a compilation of potential use cases for shapes reusage. Dec, 2024. https://doi.org/10.37044/osf.io/b4j7y
[15] Fernández-Álvarez D, García-González H, Frey J, Hellmann S, Gayo JEL. Inference of latent shape expressions associated to DBpedia ontology. In: International semantic web conference (p&d / industry); 2018.
[16] Fernández-Alvarez D, Labra-Gayo JE, Garcıa-González H. Inference and serialization of latent graph schemata using shex. In: International conference on advances in semantic processing; 2016.
[17] Fernandez-Álvarez D, Labra-Gayo JE, Gayo-Avello D. Automatic extraction of shapes using sheXer. Knowledge-Based Syst 2021:107975.
[18] Fernández-Álvarez D, Waagmeester A, Nishida K, Chaves D. WESO/sheXer. Zenodo. Jul, 2025. https://doi.org/10.5281/zenodo.15850295
[19] Fernández-Álvarez D, Yamamoto Y, Labra-Gayo JE, Waagmeester A. BioHack24 report: toward improving mechanisms for extracting RDF shapes from large inputs. 2024.
[20] Fernández-Álvarez D, Yamamoto Y, Labra Gayo JE, Waagmeester A. Extracting shapes from large RDF data collections. In: SWAT4HCLS; 2024. p. 66–75.
[21] Ghiasnezhad Omran P, Taylor K, Rodríguez Méndez S, Haller A. Learning SHACL shapes from knowledge graphs. Semantic Web 2022;14(1):101–21.
[22] Hernández D, Hogan A, Krötzsch M. Reifying RDF: what works well with wikidata? SSWS@ ISWC 2015;1457:32–47.
[23] Katayama T, Nishizawa T. RDF-config. 2015. https://github.com/dbcls/rdf-config.
[24] Keely A. SHACLgen. 2023. https://github.com/AKSW/SHACLgen.
[25] Knublauch H, Kontokostas D. Shapes constraint language (SHACL). W3C Candidate Recommendation 2017;11(8).
[26] Krech D. RDFLib: a Python library for working with RDF. 2006. https://github.com/RDFLib/rdflib.
[27] Labra-Gayo JE, Hevia AG, Álvarez DF, Ammar A, Brickley D, Gray AJG, Prud'hommeaux E, Slenter D, Solbrig H, Beghaeiraveri SAH, et al. Knowledge graphs and wikidata subsetting. 2021. https://doi.org/10.37044/osf.io/wu9et
[28] Labra-Gayo JE, Iglesias-Préstamo A, Martín-Fernández D, Arnaud MA. rudof: a rust library for handling RDF data models and shapes. In: International semantic web conference (P&D); 2024.
[29] Labra-Gayo JE, Prud'Hommeaux E, Boneva I, Kontokostas D. Validating RDF data. Morgan & Claypool Publishers; 2017.
[30] Mihindukulasooriya N, Rashid MRA, Rizzo G, García-Castro R, Corcho O, Torchiano M. RDF shape induction using knowledge base profiling. In: Proceedings of the 33rd annual ACM symposium on applied computing; 2018. p. 1952–9.
[31] Prud'hommeaux E, Baker T. Shape maps specification. 2017. https://shex.io/shape-map/.
[32] Prud'hommeaux E, Boneva I, Labra-Gayo JE, Kellogg G. Shape expressions language 2.1. W3C Community Group Report. 2019.
[33] Rabbani K, Lissandrini M, Hose K. Extraction of validating shapes from very large knowledge graphs. Proc VLDB Endow 2023;16(5):1023–32.
[34] Rabbani K, Lissandrini M, Hose K. SHACTOR: improving the quality of large-scale knowledge graphs with validating shapes. In: Companion of the 2023 international conference on management of data; 2023. p. 151–4.
[35] Seifer P, Hernández D, Lämmel R, Staab S. From shapes to shapes: inferring SHACL shapes for results of SPARQL construct queries. In: Proceedings of the ACM web conference 2024; 2024. p. 2064–74.
[36] Spahiu B, Maurino A, Palmonari M. Towards improving the quality of knowledge graphs with data-driven ontology patterns and SHACL. In: ISWC (best workshop papers); 2018. p. 103–17.
[37] Spahiu B, Porrini R, Palmonari M, Rula A, Maurino A. ABSTAT: ontology-driven linked data summaries with pattern minimalization. In: European semantic web conference. Springer; 2016. p. 381–95.
[38] Waagmeester A, Willighagen EL, Su AI, Kutmon M, Gayo JEL, Fernández-Álvarez D, Groom Q, Schaap PJ, Verhagen LM, Koehorst JJ. A protocol for adding knowledge to wikidata: aligning resources on human coronaviruses. BMC Biol 2021;19(1):1–14.
[39] Wardenga JG, Käfer T. Leveraging data shapes in large language model contexts for question answering on public and private knowledge graphs. In: TEXT2SPARQL challenge, co-located with Text2KG at ESWC; 2025. p. 1–19.
[40] Wright J, Rodríguez Méndez SJ, Haller A, Taylor K, Omran PG. Schimatos: a SHACL-based web-form generator for knowledge graph editing. In: International semantic web conference. Springer; 2020. p. 65–80.
[41] Yamamoto Y, Yamaguchi A, Splendiani A. YummyData: providing high-quality open life science data. Database 2018;2018:bay022.