Rewrite Once, Validate Anywhere: Producing OWL-Aware SHACL Constraints

Anouk Oudshoorn[1], Piotr Gorczyca[2], and Dörthe Arndt[2,3]

[1] Institute of Logic and Computation, TU Wien, Austria
[2] Computational Logic Group, TU Dresden, Germany
[3] ScaDS.AI, Dresden/Leipzig, Germany
anouk.oudshoorn@tuwien.ac.at, {piotr.gorczyca,doerthe.arndt}@tu-dresden.de

Abstract. The Shapes Constraint Language (SHACL) is a W3C recommendation to express syntactic constraints, called shapes, on RDF graphs. SHACL validators are used to test whether a given graph adheres to such a shape. However, RDF graphs often come with OWL ontologies, whose implicit knowledge needs to be taken into account. This is classically handled by first applying reasoning and then performing the constraint checking on the results, often using different technologies which makes the process inefficient and vulnerable for mistakes.
To overcome this, we propose to internalise the OWL axioms in the SHACL constraints; we construct a rewriting which takes as input both shapes and an OWL EL$^-$ ontology – a fragment of OWL EL restricting the usage of existential restrictions – and produces SHACL constraints. This output can then be evaluated by any validator supporting SHACL core regardless of its reasoning support, while yielding the same results as the traditional approach. The implementation of our translation is evaluated both against applying state-of-the-art reasoners and validators consecutively, as against validators with built-in reasoning support. For our benchmark, we show that our approach is in general more efficient in finding violations compared to the sequential approach, thus providing a powerful tool which simplifies combining reasoning with validation.

Keywords: SHACL validation · OWL reasoning · Rewriting

1 Introduction

The W3C recommendation SHACL [22] makes it possible to specify syntactic constraints on RDF graphs, which may be checked for compliance. However, RDF graphs are not only syntactical constructs but also semantic statements. That is, they use OWL DL ontologies [7] to define classes and predicates; they come with implicit knowledge. According to the specification, SHACL implementations only need to support subclass reasoning, all other reasoning support is optional [22, Section 1.5]. As a result, the SHACL validation landscape is diverse. Many systems only take the required subclass reasoning into account when performing constraint validation (e.g. Jena [4], TopBraid [20], and Isaitb [16]),

while others support OWL DL profiles (e.g., pySHACL [35]). The tasks of complex reasoning and validation are thus often executed by two independent systems, which creates a technical overhead. OWL reasoners are furthermore not always well maintained [2,24] and not all reasoning results are relevant for validation. Ke et al. [18] address the latter problem in their paper. They carefully consider which facts need to be materialised and combine this with a slight rewriting. They present an efficient way to perform SHACL validation in presence of OWL LD [11]. However, their approach requires two distinct steps for each data graph: first the RE-SHACL system needs to be run, followed by a standard SHACL validator. We consider another ontology language that is not contained in OWL LD and aim to go another route; we propose a data independent rewriting. Based solely on the ontology axioms and shapes graph, we produce a new shapes graph. This shapes graph can then be applied to any input data graph with any SHACL validator.

This is illustrated by the following example. We show a constraint which makes sure that every person has at least one name.[4]

```
:PersonShape
    a sh:NodeShape ;
    sh:targetClass :Person ;
    sh:property [
        sh:path :name ;
        sh:minCount 1 ;
    ] .
```

Moreover, we know that every student is a person, `:Student rdfs:subClassOf :Person.`. Here, we can add a target declaration `:PersonShape sh:targetClass :Student.` to make sure that the shape `:PersonShape` also targets all instances of the class `:Student`.

This does not only work for targets but also for constraints. If the ontology, for example, contains a subproperty `:studentName rdfs:subPropertyOf :name.`, we can replace the IRI `:name` (line 5) by `[ sh:alternativePath (:name :studentName) ]`, and ask for a *name* or a *student name* instead. The idea is thus that all constraint violations that could be found by first performing ontology reasoning and then executing constraint checking can also be found by only performing constraint checking using the rewritten shapes. The advantage of this approach is that the results become independent of the reasoning a SHACL validator supports and are therefore well-suited to be shared through the Web. In practical set-ups with fixed ontologies and shapes, but varying data graphs, only one system, the SHACL validator, needs to be maintained.

Note that the idea behind this form of rewriting differs from classical rewriting of SPARQL [14] queries based on OWL-QL [23] as first proposed by Poggi et al. [32] and realised in many applications [37]. When we rewrite SPARQL based on OWL QL axioms, we modify one single existing query. This does not allow for any recursion beyond SPARQL's property paths. A SHACL graph consists of several shapes which may depend on each other and while recursion between

[4] We omit the prefix declarations for brevity.

different shapes is not yet part of the standard, this extension is planned for the next version [1].

The SHACL working group furthermore plans to add rule support [9] to the specification. Engines supporting this feature will naturally be able to cope with RL using the corresponding rules [23, Section 4.3]. Being expressible in Datalog, these rules could alternatively be directly incorporated into SHACL shapes using the rewriting Pareti et al. provided [31]. We therefore do not focus on the RL fragment here.

There have been different contributions to ontology-based SHACL rewriting: Savkovic et al. provide a rewriting of positive SHACL constraints based on OWL QL axioms [33]. This got extended in Ahmetaj et al. [3] to a rewriting for OWL QL and SHACL constraints with a restricted form of negation and recursion. However, their algorithm is best-case exponential and can thus not be used in practice. Oudshoorn et al. extended the former for the description logics $\mathcal{ELHI}$ [29] and Horn-$\mathcal{ALCHIQ}$ [30], which both subsume OWL QL. They present a rewriting procedure for $\mathcal{ELHI}$ that is no longer best-case exponential (only worst-case, but given their ExpTime-completeness results, this is inevitable) and thereby provide a first step towards an implementation. However, all of these contributions stayed on a theoretical level. If we broaden our horizon beyond SHACL, there are more approaches [10,19,25,26] which discuss the theory of integrating constraints and implicit knowledge or OWL.

In this paper we make the step from theory to practice: we present ShaCOWL, a tool which rewrites SHACL shapes based on OWL ontologies to enable ontology-aware validation. We extend the existing theory by considering nominals, role chains and qualified existential restrictions on the left-hand side of the ontology axioms. However, we limit the usage of existential restrictions on the right-hand side as these are known to be the main reason for the exponential blow-ups [3,29]. The resulting fragment OWL EL$^-$ still supports non-trivial axioms and is expressive enough to be used in practical applications [13]. Identifying this fragment for which we can extract an implementable algorithm, compared to the theoretically heavy rewriting techniques presented in related work (that are best case exponential size), is our first contribution. Our second contribution is to adapt existing rewriting techniques to our setting. In particular, we describe a new approach for rewriting targets as well. We provide an implementation, and perform an extensive evaluation using different OWL reasoners and SHACL validators. We compare the execution times between two approaches: (1) we rewrite the shapes based on the ontology and then validate the data graph against the rewritten shapes graph, (2) we reason on the data graph using the ontology and then validate the result against the original shapes graph. The pure validation times are similar in both approaches. This backs up our claim that rewritten shapes can be shared and used in practical applications. Links to the implementation source code and benchmarks are provided in the supplementary material statement before the references.

2 Preliminaries

Before getting to the body of this paper, we define the following notions.

Data Graphs. Let N_C, N_R and N_I denote countably infinite, mutually disjoint sets of *concept names* (also known as *class names*), *role names* (or, *property names*), and *individuals* (or, *constants*), respectively. We assume $\{\top, \bot\} \subseteq N_C$. Let $\overline{N_R} := \{p, p^- \mid p \in N_R\}$ denote the set of *roles*. In general, we will write $p \in N_R$ and $r \in \overline{N_R}$, to denote the distinction. For every $p \in N_R$, let $(p^-)^- = p$. An *atom* (or, *assertion*) is an expression of the form $A(c)$ or $p(c, c')$, for $A \in N_C$, $p \in N_R$ and $\{c, c'\} \subseteq N_I$. A *data graph* $\mathcal{A}$ is a finite set of atoms.

An *interpretation* is a pair $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$, where $\Delta^\mathcal{I}$ is a non-empty set, called the *domain*, and $\cdot^\mathcal{I}$ is a function that maps every $A \in N_C$ to a set $A^\mathcal{I} \subseteq \Delta^\mathcal{I}$, every $p \in N_R$ to a binary relation $p^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$, and every individual $c \in N_I$ to an element $c^\mathcal{I} \in \Delta^\mathcal{I}$. Let $(p^-)^\mathcal{I} := \{(c', c) \mid (c, c') \in p^\mathcal{I}\}$. The *canonical interpretation* $\mathcal{I}_\mathcal{A}$ of a data graph $\mathcal{A}$ is defined by letting $\Delta^{\mathcal{I}_\mathcal{A}}$ contain all individuals occurring in $\mathcal{A}$, and setting $c^{\mathcal{I}_\mathcal{A}} := c$ for each $c \in N_I$, $A^{\mathcal{I}_\mathcal{A}} := \{c \mid A(c) \in \mathcal{A}\}$ for each $A \in N_C$ and $r^{\mathcal{I}_\mathcal{A}} := \{(c, c') \mid r(c, c') \in \mathcal{A}\}$ for all $r \in \overline{N_R}$. We make the standard name assumption, which means $c^\mathcal{I} = c$ for all interpretations $\mathcal{I}$, and all $c \in N_I$. Note that this enforces a unique name assumption too.

(Non-recursive) SHACL. Let N_S be an infinite set of shape names s. Based on the work of Corman et al. [8], we say a *(SHACL) constraint* $s \leftarrow \varphi$ is formed from a shape name $s \in N_S$ and a shape expression φ, defined in the following way

$$\varphi ::= c \mid s \mid A \mid \neg\varphi \mid \varphi \sqcap \varphi \mid \geq_n E.\varphi \mid E = E' \mid E \neq E',$$

where $c \in N_I$, $s \in N_S$, $A \in N_C$, $n \geq 1$, $p \in N_R$ and E and E' regular path expressions, i.e., regular expressions over the language $\overline{N_R}$, defined as

$$E ::= r \mid E \cup E \mid E \cdot E \mid E^*.$$

We use $\forall E.\varphi$ as a shorthand for $\neg\geq_1 E.\neg\varphi$, and $\varphi \sqcup \varphi'$ for $\neg(\neg\varphi \sqcap \neg\varphi')$. Note that a similar trick does not work for $E = E'$ and $E \neq E'$, as sets of atoms being equal or completely disjoined cannot be simply expressed in terms of each other by using negation. Let L_E be the language defined by some regular expression E, containing words $w \in \Sigma^*$, inductively defined as $L_r := \{r\}$, $L_{E \cup E'} := L_E \cup L_{E'}$, $L_{E \cdot E'} := \{w \cdot w' \in \Sigma^* \mid w \in L_E, w' \in L_{E'}\}$, and $L_{E^*} := \{w^* \in \Sigma^* \mid w \in L_E\}$. We say E is a regular *path* expression, when $\Sigma = \overline{N_R}$. The semantics of regular path expressions is given in terms of the evaluation $E^\mathcal{I}$ over an interpretation $\mathcal{I}$, which is defined as follows. A pair (e, e') is contained in $E^\mathcal{I}$ iff there exists $r_0 \cdots r_n \in L_E$ and $\{e_1, \ldots e_n\} \subseteq \Delta^\mathcal{I}$ such that $(e, e_1) \in r_0^\mathcal{I}$, $(e_n, e') \in r_n^\mathcal{I}$ and for all $1 \leq i \leq n-1$, $(e_i, e_{i+1}) \in r_i^\mathcal{I}$.

A *constraint set* $\mathcal{C}$ is a set of SHACL constraints such that for each $s \in N_S$, there exists at most one $s \leftarrow \varphi \in \mathcal{C}$, and such that there are no cyclic dependencies. The semantics of SHACL is defined in a recursive manner by

$$
\begin{aligned}
\mathcal{I}(c) &:= \{c^{\mathcal{I}}\} \\
\mathcal{I}(s) &:= \{e \in \varphi^{\mathcal{I}} \mid s \leftarrow \varphi \in \mathcal{C}\} \\
\mathcal{I}(A) &:= A^{\mathcal{I}} \\
\mathcal{I}(\neg\varphi) &:= \Delta^{\mathcal{I}} \setminus \mathcal{I}(\varphi) \\
\mathcal{I}(\varphi \sqcap \varphi') &:= \mathcal{I}(\varphi) \cap \mathcal{I}(\varphi') \\
\mathcal{I}(\exists_{\geq n} E.\varphi) &:= \{e \in \Delta^{\mathcal{I}} \mid |\{e' \in \Delta^{\mathcal{I}} \mid (e,e') \in E^{\mathcal{I}} \wedge e' \in \mathcal{I}(\varphi)\}| \geq n\} \\
\mathcal{I}(E = E') &:= \{e \in \Delta^{\mathcal{I}} \mid \{\exists e' \in \Delta^{\mathcal{I}} \mid (e,e') \in E^{\mathcal{I}}\} = \{e' \in \Delta^{\mathcal{I}} \mid (e,e') \in E'^{\mathcal{I}}\}\} \\
\mathcal{I}(E \neq E') &:= \{e \in \Delta^{\mathcal{I}} \mid \{\exists e' \in \Delta^{\mathcal{I}} \mid (e,e') \in E^{\mathcal{I}}\} \cap \{e' \in \Delta^{\mathcal{I}} \mid (e,e') \in E'^{\mathcal{I}}\} = \emptyset\}
\end{aligned}
$$

Fig. 1: Evaluating shape expressions

the function $\mathcal{I}(\cdot)$, given in Figure 1. Given an interpretation $\mathcal{I}$ and a shape assignment S, we say a node $c \in N_I$ *validates* a shape expression φ, when $c \in \mathcal{I}(\varphi)$. Furthermore, let $\mathcal{G}$ be a set of targets of the form $s(X)$, for

$$X ::= c \mid A \mid \exists r.\top$$

where $c \in N_I$, $A \in N_C$ and $r \in \overline{N_R}$. A pair $(\mathcal{C}, \mathcal{G})$ consisting of a constraint set and set of targets is called a *shapes graph*. Given an interpretation $\mathcal{I}$, we say $\mathcal{I}$ *validates* $(\mathcal{C}, \mathcal{G})$ if for all $s(c) \in \mathcal{G}$, we find c validates s, and for all $s(X) \in \mathcal{G}$, all nodes in $X^{\mathcal{I}}$ validate s. Considering readability, we will write $\mathcal{A}$ validates $(\mathcal{C}, \mathcal{G})$, for any set of atoms $\mathcal{A}$, in which case the canonical interpretation $\mathcal{I}_{\mathcal{A}}$ is intended.

3 Ontology Language OWL EL$^-$

For the ontology axioms, we introduce the language OWL EL$^-$, a fragment of OWL EL. This logic corresponds to the description logic $\mathcal{EL}^{++}$ [5,6], but is disallowing the usage of existential restrictions on the right-hand side. An example of an ontology expressible in this fragment in the Riskman ontology [13].

Let an OWL EL$^-$ TBox $\mathcal{T}$ be a set of axioms of the form $r \sqsubseteq p$, for $r \in \{p', p' \circ p \mid p' \in N_R\}$, or $C \sqsubseteq B$, where

$$C ::= \top \mid A \mid c \mid \mathit{ran}(p) \mid C \sqcap C \mid \exists p.C$$

where $\{A, B\} \subseteq N_C$, $c \in N_I$ and $\{p, p'\} \subseteq N_R$. Furthermore, we use $\mathit{dom}(p)$ as a shorthand for $\exists p.\top$. Any expression that may occur on the left-hand side of an OWL EL$^-$ axiom is referred to as an OWL EL$^-$ *concept*.

Note that, to avoid intractability of this logic, we reduced the syntax of axioms of the form $p' \circ p \sqsubseteq p''$ to $p \circ p \sqsubseteq p$. This is a slightly weaker form of the syntactic restriction proposed by Baader et al. [6], which ensures tractability. Note that transitivity axioms may still be expressed using the form $p \circ p \sqsubseteq p$.

The semantics of OWL EL$^-$ is, as is usual, defined in terms of interpretations $\mathcal{I}$: an axiom $C \sqsubseteq D$ is satisfied whenever $C^\mathcal{I} \subseteq D^\mathcal{I}$. To this end, the interpretation function is extended in the following way: $\top^\mathcal{I} := \Delta^\mathcal{I}$, $(C \sqcap C')^\mathcal{I} := C^\mathcal{I} \cap C'^\mathcal{I}$, $(\exists p.C)^\mathcal{I} := \{c \mid (c, c') \in p^\mathcal{I} \wedge c' \in C^\mathcal{I}\}$, $(ran(p))^\mathcal{I} := \{c \mid (c', c) \in p^\mathcal{I}\}$, and $(p' \circ p)^\mathcal{I} := \{(c, c') \mid (c, d) \in p'^\mathcal{I}, (d, c') \in p^\mathcal{I}\}$. In case all axioms in $\mathcal{T}$ are satisfied in $\mathcal{I}$, we say $\mathcal{I}$ is a *model* of $\mathcal{T}$. As is standard, we define subformulas in the following recursive way: $sub(\mathcal{T}) := \{sub(X), sub(B) \mid X \sqsubseteq B \in \mathcal{T}\}$, and $sub(C \sqcap C') := \{C \sqcap C', sub(C), sub(C')\}$, $sub(\exists p.C) := \{\exists p.C, sub(C)\}$ and $sub(Y) := \{Y\}$ for $Y \in N_C \cup N_I \cup \{\top, ran(p) \mid p \in N_R\}$.

Syntactic Restrictions. To ensure that counting to at least two on complex path expressions is restricted to simple roles, we assume the following restriction on $\mathcal{C}$ and $\mathcal{T}$ to be honoured: if $\geq_n p.\varphi$ appears somewhere in $\mathcal{C}$, there do not exist $p' \circ p \sqsubseteq p \in \mathcal{T}$ or $p' \sqsubseteq p \in \mathcal{T}$.

Moreover, in the rewriting itself we are considering *acyclic* TBoxes: for each $\mathcal{T}$, we assume that there exists no $\{C_1 \sqsubseteq D_1, \ldots, C_n \sqsubseteq D_n\} \subseteq \mathcal{T}$ such that for each pair D_i and C_{i+1}, and the pair D_n, C_1, there is a concept or role name in the intersection of the syntax of the pair.

We note that both cyclic TBoxes as recursive SHACL do not break our theory, nor the correctness of our rewriting. However, to capture the cyclicity, we would produce (stratified) recursive SHACL constraints, that would need to be interpreted under a least fixed point semantics.[5] To the best of our knowledge, the only (proof-of-concept) implementation that handles recursion in a principled way is shaWell [28]. That is, to allow for a more extensive evaluation, we stick to non-recursive SHACL and acylic TBoxes in this paper.

4 Combining SHACL and OWL

We aim to provide a feasible method to combine OWL reasoning and SHACL validation in practical applications. To this end, we first need to clarify which semantics we consider. Several papers investigated the theory of this combination [3,29,30,33], and all of them have in common that SHACL constraints are evaluated over some minimised canonical model of the original data graph and OWL constraints. These papers mention multiple ways to define such a minimal model: minimisation after Skolemisation, or using the core operation on any universal model. As we disallow existential restrictions on the right hand side of an axiom, we do not face the problem these definitions aim to solve: for all new information we possibly derive using axioms, we always know which individuals are affected. Building a unique universal model that is minimal is thus straightforward.

Definition 1 *Given a set of OWL EL$^-$ axioms $\mathcal{T}$ and a data graph $\mathcal{A}$, let $\mathcal{A}_\mathcal{T}$ be the smallest set such that $\mathcal{A} \subseteq \mathcal{A}_\mathcal{T}$ and moreover:*

[5] Note that the stable-model and well-founded semantics would also suffice, as they coincide with the least fixed point semantics for stratified constraints.

- *for each* $C \sqsubseteq B \in \mathcal{T}$*, for* C *an OWL EL*$^-$ *concept if* $c \in C^{\mathcal{I}}$*, then* $B(c) \in \mathcal{A}_{\mathcal{T}}$*;*
- *for each* $r \sqsubseteq p \in \mathcal{T}$*, for* $r \in \{p', p' \circ p \mid p' \in N_R\}$*, if* $(c, c') \in r^{\mathcal{I}}$*, then* $p(c, c') \in \mathcal{A}_{\mathcal{T}}$*,*

where $\mathcal{I}$ *is a shorthand for* $\mathcal{I}_{\mathcal{A}_{\mathcal{T}}}$*.*

The following is immediate.

Proposition 1 *Given a set of OWL EL*$^-$ *axioms* $\mathcal{T}$ *and a data graph* $\mathcal{A}$*, we find that the canonical interpretation of* $\mathcal{A}_{\mathcal{T}}$ *is a model of* $(\mathcal{T}, \mathcal{A})$*. Moreover, there does not exist a smaller model of* $(\mathcal{T}, \mathcal{A})$*: for each model* $\mathcal{J}$ *of* $(\mathcal{T}, \mathcal{A})$*, we find that* $A^{\mathcal{I}} \subseteq A^{\mathcal{J}}$ *and* $p^{\mathcal{I}} \subseteq p^{\mathcal{J}}$*, for all* $A \in N_C$ *and* $p \in N_R$*.*

This brings us to the semantics of SHACL in presence of OWL EL$^-$ axioms.

Definition 2 *Given a data graph* $\mathcal{A}$*, a set of OWL EL*$^-$ *axioms* $\mathcal{T}$ *and a shapes graph* $(\mathcal{C}, \mathcal{G})$*. We say* $(\mathcal{T}, \mathcal{A})$ *validates* $(\mathcal{C}, \mathcal{G})$ *whenever* $\mathcal{A}_{\mathcal{T}}$ *validates* $(\mathcal{C}, \mathcal{G})$*.*

The rest of this paper focuses on how we can replace this two-stage method by a more direct one. We want to build a set of SHACL constraints, based on a shapes graph $(\mathcal{C}, \mathcal{G})$ and a set of axioms $\mathcal{T}$, that can directly be validated against any data graph. We will show the following:

Theorem 1. *Given a shapes graph* $(\mathcal{C}, \mathcal{G})$ *and an OWL EL*$^-$ *TBox* $\mathcal{T}$*, there exists a shapes graph* $(\mathcal{C}_{\mathcal{T}}, \mathcal{G}_{\mathcal{T}})$ *such that for every data graph* $\mathcal{A}$*,*

$$\mathcal{A}_{\mathcal{T}} \text{ validates } (\mathcal{C}, \mathcal{G}) \text{ iff } \mathcal{A} \text{ validates } (\mathcal{C}_{\mathcal{T}}, \mathcal{G}_{\mathcal{T}}).$$

The main advantage of updating the shapes graph in this way is that for every data graph, and every update of any of these data graphs, it suffices to use $(\mathcal{C}_{\mathcal{T}}, \mathcal{G}_{\mathcal{T}})$ – which only needs to be computed once – when testing validation.

The rewriting approach consists of two key phases:

- *constraint rewriting* – integration of the axioms in $\mathcal{T}$ into the input constraints $\mathcal{C}$.
- *target rewriting* – using the constraint rewriting, fine-tuning targeting such that also targets implied by $\mathcal{T}$ are addressed.

To understand the distinction between the two aspects, consider the following example.

Example 1. Let $\mathcal{T} = \{\exists p.A \sqsubseteq B, p \sqsubseteq q\}$, $\mathcal{A} = \{A(b), p(a, b)\}$, $\mathcal{C} = \{s \leftarrow \exists q.\top\}$, $\mathcal{G} = \{s(B)\}$ be the sets of ontology axioms, assertions, shapes and targets, respectively. In this set-up, we get $\mathcal{A}_{\mathcal{T}} = \mathcal{A} \cup \{B(a), q(a, b)\}$. First, we want to internalise in the shapes graph that nodes like a conforming to $\exists p.A$ should also be considered a target for s. This part is addressed in the target rewriting. Second, since $q(a, b) \in \mathcal{A}_{\mathcal{T}} \setminus \mathcal{A}$, we would like to update the constraint set to $s \leftarrow \exists(p \cup q).\top$, such that we may indeed conclude that a validates s.

In the following section, we go into more detail of these rewriting techniques.

5 Rewriting Techniques

The main idea of the rewriting is to capture all reasoning of the axioms in SHACL constraints. The most direct translation of the axioms can be found in the $shape(X, \mathcal{T})$-part: here, it is ensured that if a node is labelled with s_C for some concept C, then C must be derivable by $\mathcal{T}$ for that specific node. That is, let $shape(X, \mathcal{T})$ be defined as

$$shape(X,\mathcal{T}) := \begin{cases} s_a \leftarrow a & \text{if } X = a \\ s_{\exists p.C} \leftarrow \exists p.s_C & \text{if } X = \exists p.C \\ s_{\exists p.\top} \leftarrow \exists p.\top & \text{if } X = dom(p) \\ s_{\exists p^-.\top} \leftarrow \exists p^-.\top & \text{if } X = ran(p) \\ s_{C_1 \sqcap \ldots \sqcap C_n} \leftarrow s_{C_1} \sqcap \ldots \sqcap s_{C_n} & \text{if } X = C_1 \sqcap \ldots \sqcap C_n \\ s_B \leftarrow B & \text{if } X = B, B \in N_C \\ \qquad \sqcup \bigsqcup_{\{p \mid ran(r) \sqsubseteq B \in \mathcal{T}\}} s_{\exists p^-.\top} & \\ \qquad \sqcup \bigsqcup_{\{p \mid dom(r) \sqsubseteq B \in \mathcal{T}\}} s_{\exists p.\top} & \\ \qquad \sqcup \bigsqcup_{\{C \mid C \sqsubseteq B \in \mathcal{T}\}} s_C & \end{cases}$$

A standard argument based on the induction on the construction of C suffices to conclude the following.

Lemma 1 *Given an OWL EL$^-$ TBox $\mathcal{T}$, let $\mathcal{C} := \bigcup_{X \in sub(\mathcal{T})} shape(X, \mathcal{T})$, then for all $C \in sub(\mathcal{T})$ and $c \in N_I$, we have $c \in \mathcal{I}_{\mathcal{A}}(s_C)$ iff $c \in C^{\mathcal{I}_{\mathcal{A}_\mathcal{T}}}$.*

This is then used in multiple ways. First of all, to update the targets of shapes according to $\mathcal{T}$, we use the so-called *bridge*-shapes: if $s(X) \in \mathcal{G}$, we add $s^{\bowtie} \leftarrow \neg s_X \sqcup s$, for $s^{\bowtie}$ a fresh shape name that universally targets all nodes but constrains only those satisfying X, ensured by an implication encoded as a disjunction. In practice, we use the class, property, and individual declarations to perform the universal targeting.[6] If the ontology contains, e.g., a triple `:r a owl:ObjectProperty.`, we add target declarations for subjects and objects of `:r` using `sh:targetSubjectOf` and `sh:targetObjectOf`. We then filter out the nodes which can be labelled by s_X. For those nodes, we ensure that s is validated, as is required. Thus, let the set of targets be updated to

$$\mathcal{G}_\mathcal{T} = \mathcal{G} \cup \left\{ s^{\bowtie}(\top) \;\middle|\; s(X) \in \mathcal{G} \right\}.$$

Next, to take care about the role inclusions, we first collect the relevant role dependencies. Here, $\rightarrow_{role}$ and $\rightarrow_{chains}$ are two relation given by $p' \rightarrow_{role} p$ iff $p' \sqsubseteq p \in \mathcal{T}$, respectively $p' \rightarrow_{chains} p$ iff $p' \circ p \sqsubseteq p \in \mathcal{T}$. With $*$ and $+$ we denote

[6] Note that tools like for example Protegé [27] automatically add such declarations.

the reflexive-transitive and transitive closure of relations. That is, let

$$\begin{aligned} \mathit{Base}(\mathcal{T},p) &:= \{p' \mid p' \to^*_{\mathit{role}} p\} \\ \mathit{Dep}(\mathcal{T},p) &:= \left\{p' \;\middle|\; p' \to^+_{\mathit{chains}} p\right\} \\ \Sigma(\mathcal{T},p) &:= \begin{cases} \bigcup_{t\,\in\,\mathit{Dep}(\mathcal{T},p)} \mathit{Base}(\mathcal{T},t) & \text{if } p\circ p \sqsubseteq p \in \mathcal{T} \\ \bigcup_{t\,\in\,\mathit{Dep}(\mathcal{T},p)} \mathit{Base}(\mathcal{T},t)\setminus\{p\} & \text{otherwise.} \end{cases} \end{aligned}$$

With these dependencies defined, we can now provide the expressions that will be substituted for occurrences of p respectively p^-.

$$\begin{aligned} L^+(\mathcal{T},p) &:= (p_1 \cup \cdots \cup p_k)^* \cdot (p'_1 \cup \cdots \cup p'_n) \\ L^-(\mathcal{T},p) &:= (p'^-_1 \cup \cdots \cup p'^-_n) \cdot (p^-_1 \cup \cdots \cup p^-_k)^* \end{aligned}$$

Here, $\{p_1,\ldots,p_k\} = \Sigma(\mathcal{T},p)$ and $\{p'_1,\ldots,p'_n\} = \mathit{Base}(\mathcal{T},p)$. To illustrate these auxiliary definitions for roles consider the following example.

Example 2. Let $\{p_0 \sqsubseteq p_1, p_1 \circ p_3 \sqsubseteq p_3, p_2 \sqsubseteq p_3\} \subseteq \mathcal{T}$, and p_3 and p_3^- appearing in shapes. Then $\mathit{Base}(\mathcal{T},p_3) = \{p_2,p_3\}$, $\mathit{Dep}(\mathcal{T},p_3) = \{p_1\}$, $\Sigma(\mathcal{T},p_3) = \{p_0,p_1\}$, and p_3 will be substituted by $(p_0 \cup p_1)^* \cdot (p_2 \cup p_3)$, whereas p_3^- by $(p_2^- \cup p_3^-) \cdot (p_0^- \cup p_1^-)^*$. Since $p_3 \circ p_3 \sqsubseteq p_3 \notin \mathcal{T}$, p_3 is excluded from $\Sigma(\mathcal{T},p_3)$ to prevent admitting repeated p_3-steps in the prefix; had p_3 been transitive, the path would expand to $(p_0 \cup p_1 \cup p_3)^* \cdot (p_2 \cup p_3)$ instead.

In the rewriting we need to substitute certain roles or concept names by fresh objects, that is, let $\varphi\big[x \mapsto y(x)\big]_{x\in X}$ and $\mathcal{C}\big[x \mapsto y(x)\big]_{x\in X}$ be the notation indicating that for each $x \in X$, every occurrence of x in φ resp. $\mathcal{C}$ is replaced by $y(x)$, where for all $p \in N_R$, we consider p^- a symbol in itself, not containing p.

Lemma 2 *Let $\mathcal{T}$ be any OWL EL$^-$ TBox only containing axioms of the form $r \sqsubseteq p$, and φ any shape expression, then we find that for each data graph $\mathcal{A}$, we have $c \in \mathcal{I}_{\mathcal{A}_\mathcal{T}}(\varphi)$ iff $c \in \mathcal{I}_\mathcal{A}(\varphi\big[p \mapsto L^+(\mathcal{T},p),\ p^- \mapsto L^-(\mathcal{T},p)\big]_{p\in N_R})$ for all $c \in N_I$.*

The main idea why this substitution suffices is that for every axiom of the form $p \sqsubseteq p'$, p' is replaced by $p \cup p'$, whereas for axioms of the form $p \circ p' \sqsubseteq p'$, p' is replaced by $p^* \cdot p'$. The sets $\Sigma(\mathcal{T},p)$ and $\mathit{Base}(\mathcal{T},p)$ solely collect the whole set of dependent p's, and do the replacements simultaneously to avoid termination issues. For the inverse substitution, dictated by L^-, exactly the same is happening, but in reverse.

For the constraint rewriting, the main idea is to substitute all concept names B appearing in the constraint set $\mathcal{C}$ by the shape name s_B, which, as mentioned before, collects exactly all nodes for which B can be derived in $\mathcal{T}$.

That is, the final rewritten set of shapes, $\mathcal{C}_\mathcal{T}$ is given by

$$\begin{aligned} \mathit{allShapes}(\mathcal{T},\mathcal{C},\mathcal{G}) := \mathcal{C}\big[B \mapsto S_B\big]_{B\,\in\,N_C} &\cup \left\{s^{\bowtie} \leftarrow \neg s_X \sqcup s \;\middle|\; s(X) \in \mathcal{G}\right\} \\ &\cup \{\mathit{shape}(X,\mathcal{T}) \mid X \in \mathit{sub}(\mathcal{T}) \cup \{X \mid s(X) \in \mathcal{G}\} \\ \mathcal{C}_\mathcal{T} := \mathit{allShapes}(\mathcal{T},\mathcal{C},\mathcal{G})\big[p \mapsto L^+(\mathcal{T},p),\ p^- &\mapsto L^-(\mathcal{T},p)\big]_{p\in N_R}. \end{aligned}$$

In a concrete setting, this produces the following constraints.

Example 3. Consider the following TBox and shapes graph

$$\mathcal{T} = \{A \sqsubseteq B,\ \exists p.B \sqsubseteq C,\ dom(p) \sqsubseteq A,\ ran(p) \sqsubseteq B,\ p_0 \sqsubseteq p,\ p_1 \circ p \sqsubseteq p,\ p_0 \sqsubseteq p_1\}$$
$$\mathcal{C} = \{s_1 \leftarrow \exists p.B, \quad s_2 \leftarrow \exists p^-.A\} \quad \mathcal{G} = \{s_1(A), \quad s_2(\exists p^-.\top)\},$$

following the rewriting as described above, this produces the following rewritten shapes graph

$$\mathcal{C}_{\mathcal{T}} = \left\{ \begin{array}{l} s_1 \leftarrow \exists\big((p_0 \cup p_1)^* \cdot (p_0 \cup p)\big).s_B, \quad s_2 \leftarrow \exists\big((p_0^- \cup p^-) \cdot (p_0^- \cup p_1^-)^*\big).s_A \\ s_A \leftarrow A \sqcup s_{\exists p.\top}, \quad s_B \leftarrow B \sqcup s_A \sqcup s_{\exists p^-.\top}, \quad s_C \leftarrow C \sqcup s_{\exists p.B} \\ s_{\exists p.\top} \leftarrow \exists\big((p_0 \cup p_1)^* \cdot (p_0 \cup p)\big).\top \\ s_{\exists p^-.\top} \leftarrow \exists\big((p_0^- \cup p^-) \cdot (p_0^- \cup p_1^-)^*\big).\top \\ s_{\exists p.B} \leftarrow \exists\big((p_0 \cup p_1)^* \cdot (p_0 \cup p)\big).s_B \\ s_1^{\bowtie} \leftarrow \neg s_A \sqcup s_1, \quad s_2^{\bowtie} \leftarrow \neg s_{\exists p^-.\top} \sqcup s_2 \end{array} \right\}$$

$$\mathcal{G}_{\mathcal{T}} = \mathcal{G} \cup \{s_1^{\bowtie}(\top), \quad s_2^{\bowtie}(\top)\}.$$

Proof of Theorem 1. The correctness of our rewriting follows from combining the intuitions of Lemmata 1 and 2 with the idea that for each $s \leftarrow \varphi \in \mathcal{C}$, with $s(X) \in \mathcal{T}$, we find that for all $c \in X^{\mathcal{I}_{\mathcal{A}_{\mathcal{T}}}}$ that c validates $s^{\bowtie}$ in $\mathcal{A}_{\mathcal{T}}$ iff c validates s in $\mathcal{A}_{\mathcal{T}}$, and for all $c \notin X^{\mathcal{I}_{\mathcal{A}_{\mathcal{T}}}}$, c validates $s^{\bowtie}$ in $\mathcal{A}_{\mathcal{T}}$ anyway, independent of whether c validates s in $\mathcal{A}_{\mathcal{T}}$ or not. □

Note that our rewriting is data independent, which means that in size of the data graph, the size of our rewriting is a constant. In size of the in general much smaller TBox, the amount of newly introduced shapes is linear. The size of rewritten constraints itself is also limited: the only blow-up in size may be caused by replacing roles by a regular expression (which size only depends on the amount of role hierarchies in the TBox).

6 Implementation and Evaluation

We implemented ShacOWL[7], a JavaScript tool realising the rewriting shown in the previous section, using the rdflib.js library (v2.2.37) for RDF parsing, graph manipulation, and Turtle serialisation. Given an OWL EL$^-$ TBox $\mathcal{T}$ and SHACL shapes $(\mathcal{C}, \mathcal{G})$, it produces rewritten shapes $(\mathcal{C}', \mathcal{G}')$ that encode the terminological knowledge directly within the constraints, eliminating the need for a reasoner while providing the same validation results. To verify its correctness and evaluate its performance we designed and carried out a set of experiments and comparisons with related systems, that will be described in this section.

Related Systems. ShacOWL offers an alternative to the baseline method, in which the datagraph is first materialised against the ontology, completing all

[7] Available at https://github.com/gorczyca/owl-aware-shacl

Approach	T.R.	Av.R.	Validator	V.E.	T.V.	Av.V.	T/O	Success (%)
Rewriting	1.05h	2.53s	IsaitB	0	1.11h	2.66s	0	1500 (100.0%)
			Jena	0	0.71h	1.71s	0	1500 (100.0%)
			pySHACL	892	0.51h	3.02s	0	608 (40.5%)
			TopBraid	0	0.81h	1.93s	0	1500 (100.0%)
HermiT	51.4h	123.3s	IsaitB	1	0.71h	1.98s	200	1299 (86.6%)
			Jena	0	0.49h	1.36s	201	1299 (86.6%)
			pySHACL	40	0.28h	0.81s	201	1259 (83.9%)
			TopBraid	1	0.49h	1.36s	200	1299 (86.6%)
JFact	158.2h	379.7s	IsaitB	0	0.32h	1.95s	902	598 (39.9%)
			Jena	1	0.17h	1.02s	902	597 (39.8%)
			pySHACL	1	0.06h	0.36s	901	598 (39.9%)
			TopBraid	0	0.20h	1.23s	902	598 (39.9%)
Pellet	43.7h	105.0s	IsaitB	0	0.72h	1.97s	185	1315 (87.7%)
			Jena	1	0.50h	1.38s	186	1313 (87.5%)
			pySHACL	47	0.21h	0.60s	186	1267 (84.5%)
			TopBraid	0	0.48h	1.31s	186	1314 (87.6%)
pySHACL	+ OWL-RL			107	14.8h	38.3s	0	1393 (92.9%)
	+ OWL-RL + RDFS			107	20.8h	53.7s	0	1393 (92.9%)

Table 1: Results per approach and validator. T.R./Av.R. – total/average reasoning or rewriting time; T.V./Av.V. – total/average validation time; V.E. – validator errors; T/O: timeouts; Success – successfully solved instances and success rate. For pySHACL with inference, reasoning and validation are a single step, so only validation statistics are reported.

the implicit class and role assertions. For the inference phase in the baseline method we used three OWL 2 reasoners: Pellet [34] (v2.4.0), HermiT [12] (v1.4.3.517), and JFact [36] (a Java port of FaCT++, v5.0.3), all integrated via the OWL API [15] (v5.1.20) through a custom JAR wrapper that returns all inferred class and object property assertions. We also considered ELK [17] (v0.6.0) and StructuralReasoner (the built-in OWL API reasoner), but excluded them as they do not support ABox completion through the OWL API, being primarily designed for other reasoning tasks such as classification. In the second phase, SHACL validation is carried out. For the ShacOWL approach this means verifying the initial datagraph against the rewritten constraints; for the baseline method it means validating the materialised datagraph against the original shapes. For validation we used four SHACL validators: Jena [4] (v4.10.0), TopBraid [20] (v1.4.3), IsaitB [16] (v1.10.0), and pySHACL [35] (v0.30.1). We additionally tested pySHACL's built-in inference feature, which accepts an input ontology alongside the data and supports RDFS, OWL RL, or combined reasoning modes. Since OWL RL subsumes OWL EL^-, we include this as a third evaluated approach. In summary, we evaluate the following approaches:

- *baseline*: materialisation with an OWL reasoner followed by SHACL validation (3 reasoners × 4 validators = 12 combinations),
- *rewriting*: rewriting with SHACOWL followed by SHACL validation (4 combinations, one per validator),
- PYSHACL *with inference*, using OWL RL or OWL RL+RDFS reasoning (2 combinations).

Benchmarks and Experiments Setup. We generated 1500 benchmarks across 15 difficulty levels (100 per level), each consisting of an OWL EL$^-$ TBox, a set of SHACL shapes, and a data graph. Difficulty scales with ontology size and complexity, varying the number of classes, roles, GCIs, role chains, role inclusions, and shape complexity. All benchmarks are acyclic; while our algorithm would still terminate on cyclic GCIs yielding recursive rewritings, current SHACL validators do not support recursive shapes. We therefore restricted evaluation to non-recursive TBoxes – an assumption that can be dropped once such validators become available. Full pseudocodes of the generation algorithms, benchmark statistics, triple counts for benchmarks, rewritings and materialised data graphs as well as and Turtle encodings of the input and output from Example 3 are provided in the appendix[8]. We note that due to the random generation process, the vast majority of instances are negative (i.e., the datagraph does not conform), with positive instances appearing almost exclusively at the lowest difficulty level. Approaches were given 1200s in total per each benchmark, split evenly between the reasoning/rewriting and validation phase, and PYSHACL+inference approach was given full 1200s for its singular task. All experiments were run on an HPC cluster, with each job allocated 64 GB of memory.

Results Analysis. The results of our experiments are presented in Figure 2 and Table 1. The first big column of the table displays the overall time (T.R) and the average time (Av.R.) of the rewriting process (first row) and of the reasoning performed by the different engines (rows 2–4). For all our test cases together the rewriting only took a little bit longer than one hour, while the fastest reasoner, Pellet, needed almost 44 hours to perform a materialisation. The reason for this difference partly lies in the complexity of the logics supported by the different systems and therefore has to be treated with some care: while our approach only supports the OWL EL$^-$ fragment, the reasoners we employed are OWL DL reasoners, which need to support more complex reasoning. As all ontologies in our benchmark fall under OWL EL$^-$, we expect the results of the reasoners to be the same an OWL EL$^-$ reasoner, if it existed, would produce. The numbers however show that rewriting can be done in a short amount of time (2.53s on average). Note that the rewriting is furthermore independent of the data graph and only needs to be executed once for each combination of ontology and shapes, while in the reasoning set-up each new data graph requires a new reasoning run.[9]

In the second big column of the table, we collected the number of validation errors (V.E), the total time (T.V) and the average time (Av.V) per validation

[8] Available through our git repository https://github.com/gorczyca/owl-aware-shacl.
[9] If only one datagraph is updated, incremental reasoners could be used.

engine. Note, that the number of validation errors is significantly higher for pySHACL than for all other engines, in particular in combination with the rewriting. The reason is that the version of pySHACL we used only supports shapes with a depth of up to 30, that is, it does not allow deeply nested shapes.[10] Such kind of nesting is rather unlikely to occur if shapes are created manually, but it is the expected output of our rewriting. If we take, for example, shape $s_1 \in \mathcal{C}$ from Example 3, and compare it to its rewritten version $s_1 \in \mathcal{C}_\mathcal{T}$, we see that the rewriting added a dependency on s_B which in turn depends on s_A and $s_{\exists r^-.\top}$ of which the former again depends on $s_{\exists r.\top}$. We thus create a chain of four dependent shapes from a single shape which had no connection to any other.[11]

For the other validation engines the average times for the rewritten shapes are higher than the validation times of the original shape on the materialised graph. The main reason for that is that this number reflects the average of all *successful* validation runs. Especially for the more complex ontologies the materialisation performed by the reasoners often failed which make these numbers difficult to compare. The numbers for the cases which could successfully evaluated by (almost) all approaches can be found in the appendix. The last column of the table reflects the failures. We display the number of successful runs of (reasoning or rewriting)+validation run (Success) and the number of runs which resulted in a time out (T/O). Only the rewriting approach was successful in all cases. The timeouts for the other approaches were all caused by the reasoners which were not able to perform materialisation in less than 600s for complex ontologies. With JFact, we could only cover around 40% of the cases while the success rate of HermiT and Pellet is around 87%. In the last row of the table, we also display the reasoning and validation time for pySHACL enabled with entailment. With this approach, we could cover around 93% of all cases.

In order to better compare all approaches to each other, we display a cactus plot in Figure 2. On the y-axis we count the number of test cases which were each solved in less than the amount of time displayed on the x-axis. The timings shown are the combined times for reasoning/rewriting and validation. We, observe that around 400 test cases could each be solved in 1 second or less with the reasoning+validation approach using Pellet and pySHACL (solid yellow line). This approach is the most successful for this short amount of time. However, for an execution time of 4s, the rewriting approach using TopBraid, Isaitb and Jena outperforms all other approaches and in less than 25s it solves all instances of the benchmark. We see that with the known exception of pySHACL, this result does not depend on the validator. This strengthens our claim that the rewritten shapes can be easily exchanged and used by different parties.

[10] pySHACL allows the user to change the maximal depth for the shapes using the option `--max-depth` [35], but this feature was broken and only got fixed with version v0.40.0 https://github.com/RDFLib/pySHACL/issues/315.

[11] Note that the example does not take SHACL's property shapes into account as these are not represented as shapes in the logical representation. The turtle version of the translated shape can be found in the appendix.

7 Discussion: towards OWL EL

Before, we considered OWL EL$^-$, a fragment of OWL EL in which existential restrictions on the right hand side of axioms are disallowed. In this section, we go over the ideas and challenges involved in converting our method into one for full OWL EL. The main issues to consider are handling the introduction of new edges to already existing nominals and the handling of the introduction of edges for which the object is not specified in the axiom. We will discuss them in this order.

Existential restrictions to nominals. When encountering axioms of the form $C \sqsubseteq \exists p.c$, this means we have to simulate jumps from nodes in the extension of the concept C to the nominal c whenever p occurs in a shape expression. Consider for instance the constraint $s \leftarrow \geq_2 p.\varphi$ with target $s(a)$. A possible way to rewrite this would be by using some kind of universal role u that connects every node to every other node:

$$s \leftarrow \geq_2 p.\varphi \sqcup (\geq_1 p.\varphi \sqcap C \sqcap \neg \geq_1 p.c \sqcap \geq_1 u.(c \wedge \varphi)).$$

However, such a universal role is in general not included in validators, meaning that we have to simulate the described setting in some way. Here, we may use that this is not a random jump through the data graph; each one of them is initiated by ontology axioms. That is, we may include the triple structure used to define the axiom $C \sqsubseteq \exists p.c$ in the rewritten shape constraint:[12] the structure $\mathit{type}(a, C)$, $\mathit{subclassOf}(C, x)$, $\mathit{someValuesFrom}(x, y)$, $\mathit{oneOf}(y, z)$, $\mathit{first}(z, c)$ might be there, in which case, $u = \mathit{type} \cdot \mathit{subclassOf} \cdot \mathit{someValuesFrom} \cdot \mathit{oneOf} \cdot \mathit{first}$ will suffice. However, this gets more complicated if a being of type C is something that will need to be derived as well.

Thus, we need to somehow move the validation process to the node c, but make it depend on the right conditions being met at the node a. Another way to achieve this would be to use Boolean combinations of targets: $s(a) \vee (s'(a) \wedge s''(c))$, where $s \leftarrow \geq_2 p.\varphi$ is the original constraint, and s' and s'' are given as follows

$$s' \leftarrow \geq_1 p.\varphi \sqcap C \sqcap \neg \geq_1 p.c \qquad\qquad s'' \leftarrow \varphi.$$

At the time of writing, there is a W3C working group developing SHACL 1.2 [21], an updated SHACL version, which among other things extends the targetting by so-called shape targets. These make it possible to target all nodes adhering to a specified shape, but they do not support boolean combinations of targets on different shapes as exemplified and needed above. That is, it remains unclear whether such an approach, although natural, will become supported in the near future.

Existential restrictions. Considering unrestricted existential restrictions in the setting of combining SHACL with reasoning has been the topic of the works by

[12] We omit prefixes for readability.

Ahmetaj et al. and Oudshoorn et al. [3,29,30], although in fragments without nominals.[13] That is, rewriting techniques for SHACL and existential restrictions are known, but also known to be ExpTime-complete in combined complexity (data graph size plus ontology size), for relatively inexpressive ontologies like OWL QL already [3,30]. The main reason for this exponential blow-up lays in having to consider every possible configuration that can appear in the data - which can only be resolved by considering which settings may actively appear in the data [29], at the cost of leaving the data independence behind. That is, it is not per se impossible to perform SHACL rewritings with unrestricted existential restrictions, but it requires care to obtain efficient rewritings.

8 Conclusion and Outlook

In this work we presented the first step from the so far rather theoretical research around ontology aware shapes rewriting towards its practical application. We identified OWL EL$^-$ as an OWL EL fragment which is expressive enough to cover non trivial use cases [13] but which can still be incorporated into SHACL shapes with rather low effort. We detailed how the rewriting can be done and provided an implementation which can be reused in other work. We performed an evaluation and compared the rewriting approach to (1) the approach performing reasoning on a data graph to then validate the shapes on the resulting materialisation using different engines and (2) to a validator with built-in reasoning. We showed that the combination of rewriting and validation could solve all our test cases and was faster than the alternatives in most cases. Moreover, our rewritten shapes can be reused for new data graphs that come with the same ontology axioms and constraints further reducing validation times. Our evaluation also revealed a potential problem, namely that the rewritten shapes get rather complex. We do not expect users to directly modify these automatically generated files. However, depending on the engine, this complexity will also be visible in the validation reports. This will make reports harder to read, but allow users to better trace the cause for a violation. Future work will help to generate extracts from such complex reports optimised for the users' needs. We furthermore want to extend our techniques to cover more expressive fragments of OWL EL by adding, for example, inverse roles. As our SHACL fragment already contains inverse roles, we expect that this generalisation could follow rather straightforwardly after a careful inspection of the described algorithms.

Acknowledgements. The project leading to this application has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101034440.

Furthermore, this work was supported by funding from BMFTR within projects SEMECO (grant no. 03ZU1210B), KIMEDS (grant no. GW0552B), and MEDGE (grant no. 16ME0529).

[13] Namely the description logics DL-Lite$_R$ (DL underlying OWL QL), $\mathcal{ELHI}$ and Horn-$\mathcal{ALCHIQ}$.

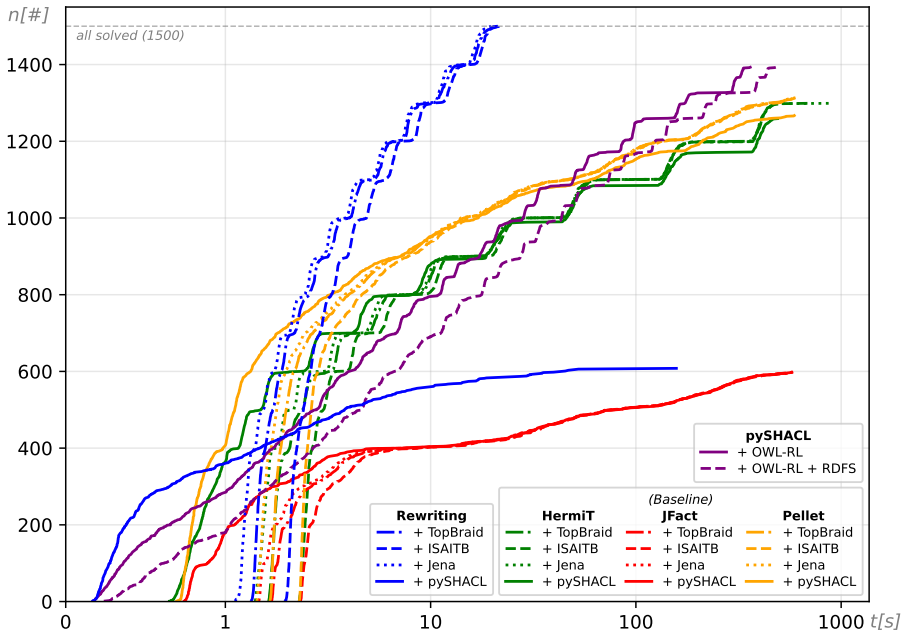

Fig. 2: Cactus plot comparing successful instances across evaluated approaches. The x-axis shows total execution time per instance in seconds (symlog scale: linear below 1s, logarithmic above). The y-axis shows the cumulative number of instances solved within that time; lines reaching higher values indicate better coverage, lines further left indicate faster execution. Colors denote the approach: blue – rewriting, green/red/orange – baseline (HermiT/JFact/Pellet), purple – pySHACL with inference. Line styles distinguish validators (TopBraid: dash-dot, Isaitb: dashed, Jena: dotted, pySHACL: solid). Rewriting (blue) achieves complete coverage of all 1500 instances in under 10s. Among the baseline methods, HermiT and Pellet reach up to 1300 instances and require up to several hundred seconds; JFact stalls at around 600 instances. Both pySHACL inference modes reach around 1400 instances and are considerably slower than the rewriting approach.

Supplemental Material Statement. The Turtle encoding of Example 3 (input and output of ShacOWL), pseudocodes of the benchmark generator, and benchmark statistics (positive-to-negative ratio, triple counts for benchmarks, materialised datagraphs, and rewritten shapes) are provided in the appendix which, together with ShacOWL's source code, is available from GitHub at https://github.com/gorczyca/owl-aware-shacl. Additional materials, including the Java OWL API wrappers for the reasoners, the benchmark generator, result summaries in CSV format, sample benchmarks for each difficulty level (1–15), scripts for setting up and running the experiments, and scripts for reproducing Table 1 and Figure 2, are available from GitHub at https://github.com/gorczyca/owl-aware-shacl-supplementary-material. The full benchmark set (1,500 instances) and raw experimental outputs are available from Figshare under a permanent DOI: https://doi.org/10.6084/m9.figshare.33169694.

Use of Generative AI. Claude (Sonnet 4.6, Anthropic) through the Claude Code VS Code extension was used to aid in the implementation of parts of the produced code, by prompting it to provide, modify, or adjust code snippets, which were then manually reviewed and modified after careful inspection. In a similar manner, Claude aided in creating scripts for running experiments on a cluster, evaluating results, and generating summaries as seen in the paper's tables and plots, all under careful supervision and with multiple reproductions. It was additionally used for minor text editing in Section 6.

References

1. Data Shapes Working Group Charter, https://www.w3.org/2024/12/data-shapes.html
2. Abicht, K.: Owl reasoners still useable in 2023 (2023), https://arxiv.org/abs/2309.06888
3. Ahmetaj, S., Ortiz, M., Oudshoorn, A., Simkus, M.: Reconciling SHACL and ontologies: Semantics and validation via rewriting. In: Gal, K., Nowé, A., Nalepa, G.J., Fairstein, R., Radulescu, R. (eds.) ECAI 2023 - 26th European Conference on Artificial Intelligence, September 30 - October 4, 2023, Kraków, Poland - Including 12th Conference on Prestigious Applications of Intelligent Systems (PAIS 2023). Frontiers in Artificial Intelligence and Applications, vol. 372, pp. 27–35. IOS Press (2023). https://doi.org/10.3233/FAIA230250, https://doi.org/10.3233/FAIA230250
4. Apache Software Foundation: Apache Jena. https://jena.apache.org (2024)
5. Baader, F., Brandt, S., Lutz, C.: Pushing the EL envelope. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005. pp. 364–369. Professional Book Center (2005), http://ijcai.org/Proceedings/05/Papers/0372.pdf
6. Baader, F., Lutz, C., Brandt, S.: Pushing the EL envelope further. In: Clark, K., Patel-Schneider, P.F. (eds.) Proceedings of the Fourth OWLED Workshop on OWL: Experiences and Directions, Washington, DC, USA, 1-2 April 2008. CEUR Workshop Proceedings, vol. 496. CEUR-WS.org (2008), https://ceur-ws.org/Vol-496/owled2008dc_paper_3.pdf

7. Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., Sattler, U., Smith, M.: Owl 2 Web Ontology Language. W3C Recommendation (Dec 2012), http://www.w3.org/TR/owl2-syntax/
8. Corman, J., Reutter, J.L., Savkovic, O.: Semantics and validation of recursive SHACL. In: Vrandecic, D., Bontcheva, K., Suárez-Figueroa, M.C., Presutti, V., Celino, I., Sabou, M., Kaffee, L., Simperl, E. (eds.) The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11136, pp. 318–336. Springer (2018). https://doi.org/10.1007/978-3-030-00671-6_19, https://doi.org/10.1007/978-3-030-00671-6_19
9. David, R., Habgood, D., Seaborne, A., Steyskal, S. (eds.): SHACL 1.2 Rules. W3C Working Draft (02 April 2026), available at https://www.w3.org/TR/shacl12-rules/
10. Fan, W., Geerts, F.: Relative information completeness. ACM Trans. Database Syst. (November 2010). https://doi.org/10.1145/1862919.1862924, http://doi.acm.org/10.1145/1862919.1862924
11. Glimm, B., Hogan, A., Krötzsch, M., Polleres, A.: OWL: yet to arrive on the web of data? In: Bizer, C., Heath, T., Berners-Lee, T., Hausenblas, M. (eds.) WWW2012 Workshop on Linked Data on the Web, Lyon, France, 16 April, 2012. CEUR Workshop Proceedings, CEUR-WS.org (2012), https://ceur-ws.org/Vol-937/ldow2012-paper-16.pdf
12. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: HermiT: An OWL 2 reasoner. Journal of Automated Reasoning **53**(3), 245–269 (2014). https://doi.org/10.1007/s10817-014-9305-1
13. Gorczyca, P., Arndt, D., Diller, M., Hampe, J., Heidenreich, G., Kettmann, P., Krötzsch, M., Mennicke, S., Rudolph, S., Straß, H.: Supporting risk management for medical devices via the riskman ontology and shapes. In: Proceedings of the 21st International Conference on Semantic Systems (SEMANTiCS 2025) (September 2025)
14. Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C recommendation, W3C (Mar 2013), https://www.w3.org/TR/2013/REC-sparql11-query-20130321/
15. Horridge, M., Bechhofer, S.: The OWL API: A Java API for OWL ontologies. Semantic Web **2**(1), 11–21 (2011). https://doi.org/10.3233/SW-2011-0025
16. Interoperability Test Bed, European Commission DIGIT: ISAITB SHACL Validator. https://github.com/ISAITB/shacl-validator (2024)
17. Kazakov, Y., Krötzsch, M., Simančík, F.: The incredible ELK: From polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies. Journal of Automated Reasoning **53**(1), 1–61 (2014). https://doi.org/10.1007/s10817-013-9296-3
18. Ke, J., Zacouris, Z.G., Acosta, M.: Efficient validation of SHACL shapes with reasoning. Proc. VLDB Endow. **17**(11), 3589–3601 (2024). https://doi.org/10.14778/3681954.3682023, https://www.vldb.org/pvldb/vol17/p3589-acosta.pdf
19. Knorr, M., Alferes, J.J., Hitzler, P.: Local closed world reasoning with description logics under the well-founded semantics. Artif. Intell. (2011). https://doi.org/10.1016/j.artint.2011.01.007, http://dx.doi.org/10.1016/j.artint.2011.01.007
20. Knublauch, H.: TopBraid SHACL API. https://github.com/TopQuadrant/shacl (2025)
21. Knublauch, H., Bergwinkl, T., Taghzouti, Y., Wright, J. (eds.): SHACL 1.2 Core. W3C Working Draft (28 April 2026), available at https://www.w3.org/TR/shacl12-core/
22. Knublauch, H., Kontokostas, D.: Shape constraint language (SHACL). W3C recommendation, W3C (Jul 2017), https://www.w3.org/TR/shacl/

23. Krötzsch, M.: OWL 2 profiles: An introduction to lightweight ontology languages. In: Eiter, T., Krennwallner, T. (eds.) Reasoning Web. Semantic Technologies for Advanced Query Answering - 8th International Summer School 2012, Vienna, Austria, September 3-8, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7487, pp. 112–183. Springer (2012). https://doi.org/10.1007/978-3-642-33158-9_4, https://doi.org/10.1007/978-3-642-33158-9_4
24. Lam, A.N., Elvesæter, B., Martin-Recuerda, F.: A performance evaluation of OWL 2 DL reasoners using ORE 2015 and very large bio ontologies. In: Proceedings of the 5th OWL Reasoner Evaluation Workshop (ORE 2015) (2015)
25. Motik, B., Horrocks, I., Sattler, U.: Adding integrity constraints to OWL. In: Golbreich, C., Kalyanpur, A., Parsia, B. (eds.) Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions, Innsbruck, Austria, June 6-7, 2007. CEUR Workshop Proceedings, vol. 258. CEUR-WS.org (2007), http://ceur-ws.org/Vol-258/paper11.pdf
26. Motik, B., Horrocks, I., Sattler, U.: Bridging the gap between OWL and relational databases. In: Proceedings of WWW. ACM (2007). https://doi.org/10.1145/1242572.1242681, https://doi.org/10.1145/1242572.1242681
27. Musen, M.A.: The protégé project: a look back and a look forward. AI Matters **1**(4), 4–12 (2015). https://doi.org/10.1145/2757001.2757003, https://doi.org/10.1145/2757001.2757003
28. Okulmus, C., Simkus, M.: SHACL validation under the well-founded semantics. In: Marquis, P., Ortiz, M., Pagnucco, M. (eds.) Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR 2024, Hanoi, Vietnam. November 2-8, 2024 (2024). https://doi.org/10.24963/KR.2024/52, https://doi.org/10.24963/kr.2024/52
29. Oudshoorn, A., Ortiz, M., Simkus, M.: Reasoning with the core chase: the case of SHACL validation over ELHI knowledge bases. In: Giordano, L., Jung, J.C., Ozaki, A. (eds.) Proceedings of the 37th International Workshop on Description Logics (DL 2024), Bergen, Norway, June 18-21, 2024. CEUR Workshop Proceedings, vol. 3739. CEUR-WS.org (2024), https://ceur-ws.org/Vol-3739/paper-7.pdf
30. Oudshoorn, A.M., Ortiz, M., Šimkus, M.: SHACL validation in the presence of ontologies: Semantics and rewriting techniques. Artificial Intelligence **352**, 104483 (2026). https://doi.org/https://doi.org/10.1016/j.artint.2026.104483, https://www.sciencedirect.com/science/article/pii/S0004370226000093
31. Pareti, P., Konstantinidis, G., Norman, T.J., Şensoy, M.: Shacl constraints with inference rules. In: Ghidini, C., Hartig, O., Maleshkova, M., Svátek, V., Cruz, I., Hogan, A., Song, J., Lefrançois, M., Gandon, F. (eds.) The Semantic Web – ISWC 2019. pp. 539–557. Springer International Publishing, Cham (2019)
32. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. In: Spaccapietra, S. (ed.) Journal on Data Semantics X. pp. 133–173. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
33. Savkovic, O., Kharlamov, E., Lamparter, S.: Validation of SHACL constraints over KGs with OWL 2 QL ontologies via rewriting. In: Hitzler, P., Fernández, M., Janowicz, K., Zaveri, A., Gray, A.J.G., López, V., Haller, A., Hammar, K. (eds.) The Semantic Web - 16th International Conference, ESWC 2019, Portorož, Slovenia, June 2-6, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11503, pp. 314–329. Springer (2019). https://doi.org/10.1007/978-3-030-21348-0_21, https://doi.org/10.1007/978-3-030-21348-0_21
34. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Journal of Web Semantics **5**(2), 51–53 (2007). https://doi.org/10.1016/j.websem.2007.03.004

35. Sommer, A., Car, N.: pySHACL. https://github.com/RDFLib/pySHACL (2021). https://doi.org/10.5281/zenodo.4750840
36. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: Proc. of IJCAR 2006. pp. 292–297 (2006). https://doi.org/10.1007/11814771_26, jFact is a Java port of FaCT++, available at https://jfact.sourceforge.net/
37. Xiao, G., Calvanese, D., Kontchakov, R., Lembo, D., Poggi, A., Rosati, R., Zakharyaschev, M.: Ontology-based data access: A survey. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18. pp. 5511–5519. International Joint Conferences on Artificial Intelligence Organization (7 2018). https://doi.org/10.24963/ijcai.2018/777, https://doi.org/10.24963/ijcai.2018/777

Appendix

Turtle Encoding. Figure 3 (Figure 4) contains the input (output) of our rewriting algorithm in Turtle format. It corresponds to the logic-based encoding from Example 3, helping to establish the relationship between these two notations. For example, the shape $s_2(\exists r^-.\top)$ corresponds to the triple `:s2 sh:targetObjectsOf :r.` (Figure 3, right, Section 8) On the other hand, the universal targetting of bridge shapes $s_1^{\bowtie}(\top)$ and $s_2^{\bowtie}(\top)$ is obtained by addressing all named individuals, instances of all class names, subjects and objects of all role names present in the ontology as shown in Figure 4, Lines 29 and 35. For this, and other computational reasons names of all ontology individuals, roles and classes have to be declared as in (Figure 3, Lines 7–13).

```
@prefix : <http://example.org/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

# Classes
:A a owl:Class .
:B a owl:Class .
:C a owl:Class .
# Object properties
:r  a owl:ObjectProperty .
:r0 a owl:ObjectProperty .
:r1 a owl:ObjectProperty .
# A ⊑ B
:A rdfs:subClassOf :B .
# ∃r.B ⊑ C
[ a owl:Restriction ;
  owl:onProperty :r ;
  owl:someValuesFrom :B ] rdfs:subClassOf :C .
# dom(r) ⊑ A
:r rdfs:domain :A .
# ran(r) ⊑ B
:r rdfs:range :B .
# r0 ⊑ r
:r0 rdfs:subPropertyOf :r .
# r1 ∘ r ⊑ r
:r owl:propertyChainAxiom ( :r1 :r ) .
# r0 ⊑ r1
:r0 rdfs:subPropertyOf :r1 .
```

```
@prefix : <http://example.org/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .

# s1 ← ∃r.B,    s1(A)
:s1 a sh:NodeShape ;
    sh:targetClass :A ;
    sh:property [
        sh:path :r ;
        sh:class :B ;
        sh:minCount 1
    ] .

# s2 ← ∃r⁻.A,    s2(∃r⁻.⊤)
:s2 a sh:NodeShape ;
    sh:targetObjectsOf :r ;
    sh:property [
        sh:path [ sh:inversePath :r ] ;
        sh:class :A ;
        sh:minCount 1
    ] .
```

Fig. 3: Input to the implementation, Turtle encoding of the example from Example 3: TBox $\mathcal{T}$ (left) and shapes $\mathcal{C}$ (right).

Benchmarks Generation. Throughout this subsection, parameter names are typeset in *teal italics* (e.g., *numClasses*) to make them easily distinguishable across the table and algorithms. Table 2 lists all parameters of the benchmark generator together with their descriptions, and provides the concrete instantiations of each parameter across the 15 difficulty levels used in the evaluation. The benchmarks are generated by three algorithms. GenerateTBox (Algorithm 1)

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix s: <https://example.org/shapes#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

# s_1 ← ∃((r_0 | r_1)* · (r_0 | r)).s_B
ex:s1 a sh:NodeShape ;
    sh:property [ sh:minCount 1 ;
            sh:node s:S_CLASS_B ;
            sh:path ( [ sh:zeroOrMorePath [ sh:alternativePath ( ex:r0 ex:r1 ) ] ] [ sh:alternativePath ( ex:r ex:r0 ) ] ) ] ;
    sh:targetClass ex:A .
# s_2 ← ∃((r_0^- | r^-) · (r_0^- | r_1^-)*).s_A
ex:s2 a sh:NodeShape ;
    sh:property [ sh:minCount 1 ;
            sh:node s:S_CLASS_A ;
            sh:path ( [ sh:alternativePath ( [ sh:inversePath ex:r ] [ sh:inversePath ex:r0 ] ) ] [ sh:zeroOrMorePath [ sh:alternativePath ( [ sh:inversePath ex:r0 ] [ sh:inversePath ex:r1 ] ) ] ] ) ] ;
    sh:targetObjectsOf ex:r .
# s_A ← A ∨ s_∃r.⊤
s:S_CLASS_A a sh:NodeShape ;
    sh:node [ sh:or ( s:S_DomainOf_r [ sh:class ex:A ] ) ] .
# s_B ← B ∨ s_A ∨ s_∃r^-.⊤
s:S_CLASS_B a sh:NodeShape ;
    sh:node [ sh:or ( s:S_CLASS_A s:S_RangeOf_r [ sh:class ex:B ] ) ] .
# s_C ← C ∨ s_∃r.B
s:S_CLASS_C a sh:NodeShape ;
    sh:node [ sh:or ( s:S_EXISTS_r_CLASS_B [ sh:class ex:C ] ) ] .
# s_1^⋈ ← ¬s_A ∨ s_1,   s_1^⋈(⊤)
s:s1_Bridge a sh:NodeShape ;
    sh:node [ sh:or ( [ sh:not [ sh:or ( s:S_CLASS_A ) ] ] ex:s1 ) ] ;
    sh:targetClass ex:A, ex:B, ex:C ;
    sh:targetObjectsOf ex:r, ex:r0, ex:r1 ;
    sh:targetSubjectsOf ex:r, ex:r0, ex:r1 .
# s_2^⋈ ← ¬s_∃r^-.⊤ ∨ s_2,   s_2^⋈(⊤)
s:s2_Bridge a sh:NodeShape ;
    sh:node [ sh:or ( [ sh:not [ sh:or ( s:s2_TargetObjectsOf_r ) ] ] ex:s2 ) ] ;
    sh:targetClass ex:A, ex:B, ex:C ;
    sh:targetObjectsOf ex:r, ex:r0, ex:r1 ;
    sh:targetSubjectsOf ex:r, ex:r0, ex:r1 .
# s_∃r.⊤ ← ∃((r_0 | r_1)* · (r_0 | r)).⊤
s:S_DomainOf_r a sh:NodeShape ;
    sh:property [ sh:minCount 1 ;
            sh:path ( [ sh:zeroOrMorePath [ sh:alternativePath ( ex:r0 ex:r1 ) ] ] [ sh:alternativePath ( ex:r ex:r0 ) ] ) ] .
# s_∃r^-.⊤ ← ∃((r_0^- | r^-) · (r_0^- | r_1^-)*).⊤
s:S_RangeOf_r a sh:NodeShape ;
    sh:property [ sh:minCount 1 ;
            sh:path ( [ sh:alternativePath ( [ sh:inversePath ex:r ] [ sh:inversePath ex:r0 ] ) ] [ sh:zeroOrMorePath [ sh:alternativePath ( [ sh:inversePath ex:r0 ] [ sh:inversePath ex:r1 ] ) ] ] ) ] .
# s_∃r.B ← ∃((r_0 | r_1)* · (r_0 | r)).s_B
s:S_EXISTS_r_CLASS_B a sh:NodeShape ;
    sh:property [ sh:path ( [ sh:zeroOrMorePath [ sh:alternativePath ( ex:r0 ex:r1 ) ] ] [ sh:alternativePath ( ex:r ex:r0 ) ] ) ;
            sh:qualifiedMinCount 1 ;
            sh:qualifiedValueShape s:S_CLASS_B ] .
# helper shape for target of s_2, evaluates to the same as s_∃r^-.⊤, because objects of r is the same as range of r
s:s2_TargetObjectsOf_r a sh:NodeShape ;
    sh:property [ sh:minCount 1 ;
            sh:path ( [ sh:alternativePath ( [ sh:inversePath ex:r ] [ sh:inversePath ex:r0 ] ) ] [ sh:zeroOrMorePath [ sh:alternativePath ( [ sh:inversePath ex:r0 ] [ sh:inversePath ex:r1 ] ) ] ] ) ] .
```

Fig. 4: Turtle output of the implementation for the example from Example 3, given the input from Figure 3.

produces an OWL EL$^-$ TBox $\mathcal{T}$ along with the sets of class names N_C, role names N_R, and two set of individuals N_I, N_A. GenerateDataGraph (Algorithm 2) populates a data graph $\mathcal{A}$ with random class and role assertions over the data graph individuals N_A, class names N_C, and role names N_R. GenerateSHACL (Algorithm 3) produces a set of SHACL shapes $\mathcal{C}$ with targets $\mathcal{G}$, using randomly generated property paths via the auxiliary GeneratePath function; as *pathContinueProbability* increases with the difficulty level, higher levels yield more deeply nested and structurally complex paths. Each difficulty level is instantiated with 100 benchmark instances, yielding 1,500 instances in total.

Benchmarks Statistics. Table 3 reports statistics for each of the 15 difficulty levels across the 1,500 generated instances. Since both the ontology and the shapes are randomly generated, it is unlikely that the data will conform to the shapes, and indeed most instances are non-conforming (negative). Level 1 is an exception with an exactly even 50/50 split; from level 2 onward virtually all instances are negative. The table also reports mean RDF triple counts per instance for the input TBox (`owl.ttl`), the input SHACL shapes (`shacl.ttl`), and the rewritten output (`rewriting.ttl`), to help get an idea of how large the output rewriting becomes with respect to the input ontology and shapes. As expected, triple counts grow with the difficulty level.

Extra Results. Table 4 complements Table 1 by reporting, for the same full instance set, the minimum, median, and maximum reasoning/rewriting and validation time per approach and validator, computed over successfully completed instances only. Table 5 shows evaluation results restricted to instances solved by every approach, but excluding JFact for the baseline method and pySHACL for rewriting, since both are unable to solve too many instances. Table 6 reports the corresponding minimum, median, and maximum timing statistics for this same restricted instance set.

Parameter	Description
numClasses (nC)	Number of named classes (class names)
numRoles (nR)	Number of object properties (role names)
numIndividuals (nI)	Total individuals; first 20% are nominals (N_I), rest are data graph individuals (N_A)
numGCIs (nGCI)	Number of general concept inclusions
numDomains (nD)	Number of domain axioms $dom(r) \sqsubseteq C$
numRanges (nRn)	Number of range axioms $ranr \sqsubseteq C$
numRoleChains (nCh)	Number of role chain axioms $r_i \circ r_j \sqsubseteq r_j$
numRoleInclusions (nInc)	Number of role inclusion axioms $r_i \sqsubseteq r_j$
avgClassAssertions (aCA)	Average class assertions per individual in the data graph
avgRoleAssertions (aRA)	Average role assertions per individual in the data graph
numShapes (nS)	Number of SHACL shapes to generate
pathContinueProbability (pCP)	Probability of extending a property path recursively

Level	nC	nR	nI	nGCI	nD	nRn	nCh	nInc	aCA	aRA	nS	pCP
1	4	2	10	4	2	2	1	1	2.5	3.0	2	0.30
2	11	5	20	10	2	2	1	1	2.5	3.1	2	0.31
3	17	8	35	22	2	2	1	2	2.5	3.2	2	0.32
4	24	12	50	40	2	2	2	4	2.5	3.5	2	0.35
5	50	25	100	80	5	5	4	8	3.0	4.0	3	0.40
6	80	40	150	130	8	8	6	12	3.0	4.5	4	0.45
7	120	60	250	200	12	12	8	18	3.5	5.0	4	0.50
8	180	90	400	300	18	18	12	25	3.5	5.5	5	0.55
9	250	125	600	450	25	25	16	35	4.0	6.0	5	0.60
10	350	175	900	650	35	35	22	50	4.0	6.5	6	0.65
11	500	250	1,300	950	50	50	30	70	4.5	7.0	7	0.65
12	700	350	2,000	1,400	70	70	40	100	5.0	8.0	8	0.70
13	1,000	500	3,000	2,000	100	100	55	140	5.0	9.0	9	0.70
14	1,500	750	4,500	3,000	150	150	75	200	5.5	10.0	10	0.75
15	2,200	1,100	6,500	4,500	220	220	100	280	6.0	11.0	11	0.75

Table 2: Benchmark generator parameters (top) and their instantiations across difficulty levels 1–15 (bottom).

Algorithm 1: GenerateTBox

Input: *numClasses*, *numRoles*, *numGCIs*, *numDomains*, *numRanges*, *numRoleChains*, *numRoleInclusions*, *numIndividuals*

Output: OWL EL$^-$ TBox $\mathcal{T}$; sets N_C, N_R, N_I, N_A

$N_C \leftarrow \{C_1, \ldots, C_{numClasses}\}$ with index $idx(C_i) = i$
$N_R \leftarrow \{r_1, \ldots, r_{numRoles}\}$ with index $idx(r_i) = i$
$N_I' \leftarrow \{a_1, \ldots, a_{numIndividuals}\}$
$N_I \leftarrow$ first $\lfloor 0.2 \cdot |N_I'| \rfloor$ elements of N_I'
$N_A \leftarrow N_I' \setminus N_I$
$\mathcal{T} \leftarrow \emptyset$

`// GCI generation`
$n \leftarrow 0$
while $n <$ *numGCIs* **do**
  $L \leftarrow$ random LHS from $\{A,\ A \sqcap B,\ \exists r.A,\ A \sqcap \exists r.B,\ \exists r.\{a\},\ A \sqcap \exists r.\{a\}\}$ where $A, B \in N_C$, $r \in N_R$, $a \in N_I$
  $B \leftarrow$ `random`(N_C)
  if *$idx(C) < idx(B)$ for every class C appearing in L* **then**
    add $L \sqsubseteq B$ to $\mathcal{T}$
    $n \leftarrow n + 1$

`// Domain/range axioms`
foreach *r in numDomains randomly chosen elements of N_R* **do**
  add $dom(r) \sqsubseteq$ `random`$(\mathcal{C})$ to $\mathcal{T}$
foreach *r in numRanges randomly chosen elements of N_R* **do**
  add $rang(r) \sqsubseteq$ `random`$(\mathcal{C})$ to $\mathcal{T}$

`// Role chain axioms`
$n \leftarrow 0$
while $n <$ *numRoleChains* **do**
  pick $r_i, r_j \in N_R$ randomly
  if $i < j$ **then**
    add $r_i \circ r_j \sqsubseteq r_j$ to $\mathcal{T}$
    $n \leftarrow n + 1$

`// Role inclusions`
$n \leftarrow 0$
while $n <$ *numRoleInclusions* **do**
  pick $r_i, r_j \in N_R$ randomly
  if $i < j$ **then**
    add $r_i \sqsubseteq r_j$ to $\mathcal{T}$
    $n \leftarrow n + 1$

return $\mathcal{T}$, N_C, N_R, N_I, N_A

Algorithm 2: GenerateDataGraph

Input: *numIndividuals*, *avgClassAssertions*, *avgRoleAssertions*
Output: Data Graph $\mathcal{A}$

$\mathcal{A} \leftarrow \emptyset$
// Class assertions
$n \leftarrow 0$
while $n < \lfloor numIndividuals \cdot avgClassAssertions \rfloor$ **do**
  $C \leftarrow \texttt{random}(N_C)$
  $i \leftarrow \texttt{random}(N_A)$
  add $C(i)$ to $\mathcal{A}$
  $n \leftarrow n + 1$
// Role assertions
$n \leftarrow 0$
while $n < \lfloor numIndividuals \cdot avgRoleAssertions \rfloor$ **do**
  $r \leftarrow \texttt{random}(N_R$
  $i \leftarrow \texttt{random}(N_A)$
  $j \leftarrow \texttt{random}(N_A)$
  add $r(i, j)$ to $\mathcal{A}$
  $n \leftarrow n + 1$
return $\mathcal{A}$

Algorithm 3: GenerateSHACL

Input: *numShapes*, *pathContinueProbability*, N_C, N_R, N_A
Output: SHACL shape set $\mathcal{C}$, target set $\mathcal{G}$

$\mathcal{C} \leftarrow \emptyset$
$\mathcal{G} \leftarrow \emptyset$
$n \leftarrow 0$
while $n <$ *numShapes* **do**
  kind $\leftarrow$ `random({class, node, subjects, objects})`
  path $\leftarrow$ `GeneratePath()`
  if *kind* $=$ `class` **then**
    $C \leftarrow$ `random`(N_C)
    $C' \leftarrow$ `random`(N_C)
    $s \leftarrow \exists\, path.\, C'$
    add s to $\mathcal{C}$
    add $s(C)$ to $\mathcal{G}$
  if *kind* $=$ `node` **then**
    $i \leftarrow$ `random`(N_A)
    $s \leftarrow \exists\, path.\top$
    add s to $\mathcal{C}$
    add $s(i)$ to $\mathcal{G}$
  if *kind* $=$ `subjects` **then**
    $r \leftarrow$ `random`(N_R)
    $s \leftarrow \exists\, path.\top$
    add s to $\mathcal{C}$
    add $s(\exists r.\top)$ to $\mathcal{G}$
  if *kind* $=$ `objects` **then**
    $r \leftarrow$ `random`(N_R)
    $C \leftarrow$ `random`(N_C)
    $s \leftarrow C$
    add s to $\mathcal{C}$
    add $s(\exists r^-.\top)$ to $\mathcal{G}$
  $n \leftarrow n + 1$
return $\mathcal{C}$, $\mathcal{G}$
Function `GeneratePath()`:
  kind $\leftarrow$ `random({simple, sequence, alternative, inverse})`
  pick $p \in [0, 1]$ uniformly
  if $p >$ *pathContinueProbability* **or** *kind* $=$ `simple` **then**
    $r \leftarrow$ `random`(N_R)
    return r
  if *kind* $=$ `sequence` **then return** `GeneratePath()` $\cdot$ `GeneratePath()`
  if *kind* $=$ `alternative` **then return** `GeneratePath()` | `GeneratePath()`
  if *kind* $=$ `inverse` **then return** `GeneratePath()`$^-$

Level	#instances		mean #triples per instance						
	Pos.	Neg.	TBox	Shapes	Data Graph	Rewriting	HermiT	JFact	Pellet
1	50	50	44	12	55	196	135	135	134
2	4	96	97	12	112	302	242	242	242
3	0	100	193	12	199	519	420	419	421
4	2	98	333	12	300	849	669	664	678
5	0	100	666	19	700	1 789	1 498	1 490	1 505
6	0	100	1 083	29	1 125	3 022	2 380	2 367	2 383
7	0	100	1 654	30	2 125	4 567	4 370	4 514	4 370
8	0	100	2 498	45	3 600	7 082	7 267	–	7 271
9	0	100	3 681	45	6 000	10 230	12 049	–	12 051
10	0	100	5 299	70	9 450	15 396	19 083	–	19 084
11	0	100	7 699	86	14 950	23 130	29 894	–	29 896
12	0	100	11 315	106	26 000	35 117	51 491	–	51 523
13	0	100	16 146	128	42 000	52 280	83 256	–	83 252
14	0	100	24 168	372	69 750	80 679	–	–	135 196
15	0	100	36 054	340	110 500	122 659	–	–	–
Total	56	1 444	7 395	88	19 124	23 854	–	–	–

Table 3: Per-level instance statistics and mean RDF triple counts. Positive/Negative: conforming/non-conforming instances. OWL: input TBox; SHACL: input shapes; Rewriting: output of the OWL-aware rewriting; HermiT/JFact/Pellet: materialized data graph after reasoning.

Approach	Min.	Med.	Max.	Validator	Min.	Med.	Max.
Rewrit.	0.18s	0.88s	15.6s	IsaitB	1.75s	2.13s	8.52s
				Jena	0.93s	1.26s	6.92s
				pySHACL	0.24s	0.48s	145.0s
				TopBraid	1.14s	1.40s	8.35s
HermiT	0.47s	2.29s	511.6s	IsaitB	1.78s	1.92s	7.84s
				Jena	0.92s	1.04s	317.8s
				pySHACL	0.29s	0.39s	270.3s
				TopBraid	1.14s	1.27s	41.4s
JFact	0.52s	1.40s	573.1s	IsaitB	1.78s	1.83s	42.0s
				Jena	0.92s	0.99s	4.24s
				pySHACL	0.29s	0.32s	0.82s
				TopBraid	1.14s	1.19s	4.61s
Pellet	0.49s	1.17s	599.4s	IsaitB	1.78s	1.89s	6.21s
				Jena	0.92s	1.05s	306.0s
				pySHACL	0.29s	0.38s	3.01s
				TopBraid	1.14s	1.23s	7.26s
pySHACL	+ OWL-RL				0.42s	6.51s	360.7s
	+ OWL-RL + RDFS				0.47s	10.9s	480.3s

Table 4: Minimum, median, and maximum reasoning/rewriting time (Min./Med./Max., left) and validation time (Min./Med./Max., right) per approach and validator, computed over successfully completed instances only (i.e., excluding reasoning/rewriting timeouts and validator errors). Note that the minimum validation times are nearly identical between validators. This is likely due to the overhead of the validator itself per call, i.e., a fixed cost of launching the validator as a fresh process. The minima do in fact differ slightly between validators (by a few milliseconds), but this difference is lost due to rounding.

Approach	T.R.	Av.R.	Val.	V.E.	T.V.	Av.V.	T/O	Succ. (%)
Rewrit.	0.44h	1.26s	IsaitB	0	0.78h	2.25s	0	1254 (100.0%)
			Jena	0	0.47h	1.36s	0	1254 (100.0%)
			TopBraid	0	0.54h	1.54s	0	1254 (100.0%)
HermiT	15.4h	44.3s	IsaitB	0	0.69h	1.97s	0	1254 (100.0%)
			Jena	0	0.47h	1.36s	0	1254 (100.0%)
			pySHACL	0	0.28h	0.81s	0	1254 (100.0%)
			TopBraid	0	0.47h	1.36s	0	1254 (100.0%)
Pellet	9.08h	26.1s	IsaitB	0	0.68h	1.95s	0	1254 (100.0%)
			Jena	0	0.47h	1.36s	0	1254 (100.0%)
			pySHACL	0	0.20h	0.59s	0	1254 (100.0%)
			TopBraid	0	0.45h	1.28s	0	1254 (100.0%)
pySHACL	+ OWL-RL			0	5.80h	16.6s	0	1254 (100.0%)
	+ OWL-RL + RDFS			0	8.77h	25.2s	0	1254 (100.0%)

Table 5: Results per approach and validator, restricted to instances solved by every approach excluding JFact for the baseline method and pySHACL for rewriting.

Approach	Min.	Med.	Max.	Validator	Min.	Med.	Max.
Rewrit.	0.18s	0.62s	6.80s	IsaitB	1.75s	2.02s	4.92s
				Jena	0.93s	1.17s	3.61s
				TopBraid	1.14s	1.32s	4.38s
HermiT	0.47s	2.22s	511.6s	IsaitB	1.78s	1.92s	7.84s
				Jena	0.92s	1.03s	317.8s
				pySHACL	0.29s	0.39s	270.3s
				TopBraid	1.14s	1.26s	41.4s
Pellet	0.49s	1.03s	432.7s	IsaitB	1.78s	1.88s	6.21s
				Jena	0.92s	1.04s	306.0s
				pySHACL	0.29s	0.37s	2.47s
				TopBraid	1.14s	1.22s	4.79s
pySHACL	+ OWL-RL				0.42s	4.59s	112.5s
	+ OWL-RL + RDFS				0.47s	7.47s	165.0s

Table 6: Minimum, median, and maximum reasoning/rewriting and validation time per approach and validator, restricted to the same instance set as Table 5 (solved by every approach, excluding JFact for the baseline method and pySHACL for rewriting), computed over successfully completed instances only.