SHACL Validation under the Well-founded Semantics

Cem Okulmus[1] , **Mantas Šimkus**[1,2]
[1]Department of Computing Science, Umeå University, Sweden
[2]Institute of Logic and Computation, TU Wien, Austria
okulmus@cs.umu.se, simkus@dbai.tuwien.ac.at

Abstract

W3C has recently introduced SHACL as a new standard for integrity constraints on graph-structured data (specifically, on RDF data). Unfortunately, the standard defines the semantics of *non-recursive* constraints only, leaving the case of recursive constraints open. This led to recent efforts into finding a suitable, mathematically crisp semantics for constraints with cyclic dependencies. In this paper, we argue that recursive SHACL can be naturally equipped with a semantics inspired in the *well-founded semantics* for recursive logic programs with default negation. This semantics is not only intuitive, but it is also computationally tractable, unlike the main previous proposals. The semantics is tolerant to constraint violations that are outside the realm of the so-called *validation targets*, which is a quality that is highly relevant in practice. In addition to defining the well-founded semantics for SHACL using a notion of *unfounded sets*, we draw a connection to the classic definition of the well-founded semantics in logic programming: we provide a simple (yet inefficient) translation of recursive SHACL under the well-founded semantics into propositional logic programs under the well-founded semantics. This provides a basis for an optimized validation engine presented in the paper: our system performs graph validation by producing an optimized logic program that can be evaluated using a deductive database engine. The system has pay-as-you-go behavior: for validation with non-recursive constraints, the system avoids using a deductive database and instead only uses SPARQL queries over an RDF triplestore.

1 Introduction

Graph-structured data is becoming increasingly popular, mainly because it does not require the development of rigid database schemas. This is important in areas where data has complex structure, is highly incomplete, or where its structure evolves over time. These features make the development of a fixed database schema difficult, driving research into topics like *Knowledge Graphs (KGs)*, *Property Graphs*, *Graph Databases*, and *RDF graphs*. Despite the spirit of flexibility in graph-structured data, the quality and structural integrity of data are still of key importance. In traditional relational databases, the so-called *integrity constraints* provide a means to enforce a strict structure and maintain quality of stored information, and we would like to have similar tools for graph-structure data. However, this is not easy: we need to strike a good balance between the power to assert some control over the structure of data, and being relaxed enough to not negate the benefits of the graph-based data model.

To address the above issue, W3C has introduced SHACL (Knublauch and Kontokostas 2017) as a new standard for expressing constraints on RDF graphs (we refer the reader to (Pareti and Konstantinidis 2021) for an excellent tutorial). Constraints in SHACL are specified using validation rules, which have features reminiscent of logic programming and concept expressions in *Description Logics* (Baader et al. 2003; Bogaerts, Jakubowski, and Van den Bussche 2022b). Unfortunately, the SHACL standard only defines the semantics of *non-recursive* SHACL constraints. The case of cyclic dependencies, or *recursion*, was intentionally left unspecified: it is currently up to the developers of SHACL validators to come up with ad-hoc solutions to handle recursion. This has led to recent efforts into finding a suitable semantics for recursive SHACL constraints: e.g, Corman et al. have introduced a semantics that is related to the notion of *supported models* known in logic programs (Corman, Reutter, and Savkovic 2018), while Andreşel et al. have explored a semantics based on *stable models* known in Answer Set Programming (Andreşel et al. 2020).

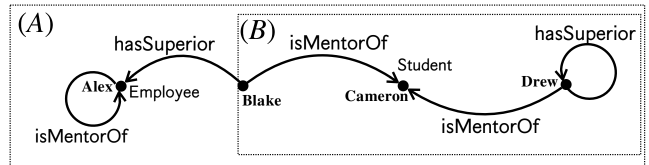
The supported model semantics of Corman et al. has two weaknesses: (a) the problem of deciding graph validity is intractable, and (b) it allows for *unfounded* (self-supported) justifications of inferences. The latter may lead to counter-intuitive validation results, as unfounded justifications may allow one to validate graphs where we expect non-validation. The stable model semantics of Andreşel et al. solves (b) but still has the problem (a): the intractability (specifically, NP-hardness) of validation in the two semantics is caused by the simultaneous presence of recursion and negation in constraints. NP-hardness already applies when the size of input constraints is assumed to be bounded by a constant (i.e., in *data complexity*), which becomes a major challenge for implementing validators for recursive SHACL since data graphs in practice are often very big. Interestingly, for the supported model semantics, NP-hardness holds already for constraints with *stratified* negation.

An important aspect of SHACL is that validation is seen as a *goal-oriented process*. In addition to the validation rules, a SHACL *document* also describes *validation targets*, whose purpose is to limit the scope of the validation process. Specifically, validation is expected to not fail merely due to

a problem in the data that is not related to the validation targets. The basic supported model and stable model semantics of Corman et al. and Andreşel et al. are 2-valued and thus do not achieve the expected resilience to "outside" problems. For this reason, Corman et al. introduced a 3-valued version of the supported model semantics (via the so-called *faithful shape assignments*), which however still suffers from unfounded justifications and intractability. Andreşel et al. also address this issue, suggesting a 3-valued stable model semantics as a way to achieve tolerance to problems outside the scope of validation targets. These authors also connect what they call *cautious validation* under the 3-valued stable model semantics for SHACL to the task of computing the *well-founded model* of a logic program with default negation (Van Gelder, Ross, and Schlipf 1991). This connection provided the first and clear hint that the main weaknesses of the previous proposals could be overcome by equipping SHACL with a semantics based on the well-founded semantics of non-monotonic logic programs.

The main goal of this paper is to provide an explicit definition of a *well-founded semantics for SHACL*, which we do using a suitable adaptation of the notion of *unfounded sets*. We believe this semantics provides a strong alternative to the previous semantics for recursive SHACL: it is intuitive, it has solid basis in logic programming, it is computationally tractable, and it can be efficiently implemented using existing tools. Specifically, our contributions are as follows:

- We provide a detailed motivating example to illustrate the basic ideas and the challenge addressed in this paper. Specifically, we illustrate the syntax of SHACL constraints and discuss some of the key problems of the previously introduced supported model semantics and the stable model semantics.
- We present a well-founded semantics for SHACL, by providing a definition based on a notion of *unfounded sets* (adapted from the notion used for logic programs). This notion avoids unfounded justifications and is tractable (not only in data complexity, but also in *combined complexity*), which resolves the two main weaknesses of the semantics in (Corman, Reutter, and Savkovic 2018). The well-founded semantics is 3-valued, which allows it to elegantly ignore data and constraint inconsistencies that are not relevant to the validation targets.
- We connect the introduced semantics for SHACL with the well-founded semantics for logic programs. To this end, we first observe that the latter can be seen as special case of the former. On the other hand, we show how the former can be encoded in a propositional logic program.
- We implemented a SHACL validator that is based on an optimized encoding to propositional logic programs. Our method—in a similar fashion to the production of a SAT instance in (Corman et al. 2019)—uses SPARQL queries to produce the desired logic program. The evaluation of this program is performed via the DLV system (Leone et al. 2006). We present our prototype validator called ShaWell and perform an experimental evaluation to compare it against other validators that support recursive SHACL.



$$\mathit{ProfShape} \leftarrow \mathit{EmplShape} \wedge (\exists \mathsf{isMentorOf}.\mathit{StudShape})$$
$$\mathit{EmplShape} \leftarrow \mathsf{Employee} \vee \exists \mathsf{hasSuperior}.\mathit{EmplShape}$$
$$\mathit{StudShape} \leftarrow \mathsf{Student} \vee (\mathit{EmplShape} \wedge \neg \mathit{ProfShape})$$

Figure 1: Example knowledge graph and SHACL constraints

2 Motivating Examples

In this section, we illustrate the main concepts related to validation of SHACL constraints over graph-structured data. To this end, we use two small knowledge graphs as follows.

Example 1 (Knowledge Graphs)**.** *See the example KG (A) in Figure 1, containing some information about members of an educational institution. The KG mentions four persons called* Alex*,* Blake*,* Cameron*, and* Drew*, who correspond to the four nodes of the KG.* Alex *is (explicitly) stated to be an employee in this institution, which is indicated by their membership in the class* Employee*.* Cameron *is a (regular) student in this institution, which is indicated by their membership in the class* Student*. The properties* isMentorOf *and* hasSuperior *connect some of the nodes in the graph, carrying the obvious meaning. The KG (B) is obtained from (A) by dropping the node* Alex*, i.e., deleting the class membership and property assertions that involve* Alex*.*

The KGs in Figure 1 contain only raw facts, without schema information or integrity constraints. To address this, SHACL provides a language for describing nodes in a KG and placing constraints on them. This is done using two components: *validation rules* and *validation targets*.

Example 2 (Validation rules)**.** *Consider the three expressions at the bottom of Figure 1, which correspond to three* validation rules *to identify nodes of three types: nodes corresponding to professors, employees, and students, respectively. Each of these three types has a name (i.e., ProfShape, EmplShape, and StudShape) that is associated to its* definition *using a possibly complex expression (similar to a concept expression in Description Logics). Specifically, the first rule tells us that a person is considered to be a professor (has type ProfShape) if they are an employee (of type EmplShape) and they mentor at least one student (an object with type StudShape). According to the second rule, a person is an employee (has type EmplShape) if that is stated explicitly (via membership in the class* Employee*), or the person has a superior that is in turn of type EmplShape. The last rule tells us that an object is of type StudShape if the object is explicitly stated to be an instance of the class* Student*, or the object is of type EmplShape but not of type ProfShape. Intuitively, in our example, the employees of the institution that are not professors can be seen as students who can, e.g., take courses.*

Note that in the view of logic programming and deductive databases, the symbols $\mathit{ProfShape}$, $\mathit{EmplShape}$, $\mathit{StudShape}$ are *monadic intensional* predicates, whereas Employee and Student are *unary extensional* predicates, while isMentorOf and hasSuperior are *binary extensional* predicates.

SHACL documents contain not only validation rules but also a specification of *target nodes*. In the basic case, we can list pairs of concrete nodes and the expected types (*shape names*). For our running example, we will use one such target specification with the obvious meaning:

$$\mathcal{T}_1 = \{(\text{Blake}, \mathit{ProfShape}), (\text{Cameron}, \mathit{StudShape})\}$$

In practice, we may want more succinct representations of targets. To this end, SHACL supports the so-called *class* and *property (domain and range) targets*. E.g., with $\mathcal{T} = \{(\mathsf{Student}, \mathit{EmplShape})\}$ we ask each instance of Student to satisfy $\mathit{EmplShape}$ (this requirement will clearly be violated in our KG).

Intuitively, our KG (A) with the three validation rules in Figure 1 does *validate* the targets in $\mathcal{T}_1$. Indeed, Cameron satisfies $\mathit{StudShape}$ because they are explicitly stated as an instance of Student. Since Blake has Alex as a superior and Alex is explicitly stated to be an employee, Blake satisfies $\mathit{EmplShape}$. Given that Blake mentors Cameron, we infer that Blake satisfies $\mathit{ProfShape}$. Unsurprisingly, we can have targets that we don't expect to be validated by our KG and the above validation rules (under a reasonable validation semantics): this holds, e.g., for $\mathcal{T} = \{(\text{Blake}, \mathit{StudShape})\}$ and $\mathcal{T} = \{(\text{Cameron}, \mathit{ProfShape})\}$.

We next illustrate two drawbacks of the supported model semantics; one of those also applies to the stable model semantics. Those issues are addressed by the well-founded semantics for SHACL that we discuss in the next sections.

The first example illustrates the need for a 3-valued semantics, since the 2-valued supported model semantics has a significant disadvantage: if we have some issue in the graph that bars the existence of a *total shape assignment*, then the validation fails for all target specifications, even those without any connection to the problematic part of the KG.

Example 3. *According to the 2-valued supported model semantics, we must find an assignment of shape names to nodes such that: (i) a node o is a assigned a shape name s iff o satisfies the shape expression associated to s, and (ii) the target specification is respected. Unfortunately, under this semantics, the target $\mathcal{T}_1$ above cannot be validated by our KG (A), which is unexpected. The problem here is that* Alex *rules out the existence of* any *shape assignment as required for validation. If we assign ProfShape to* Alex*, then* Alex *must be a mentor of some object with StudShape. Since* Alex *only mentors themselves, and StudShape is incompatible with ProfShape, our candidate assignment fails. If we decide to not assign ProfShape to* Alex*, then* Alex *must not mentor any object with StudShape. However,* Alex *mentors themselves and obviously satisfies StudShape. Thus again the shape assignment has failed.*

The 2-valued stable model semantics of Andreşel *et al.* has the same problem as above. To overcome this issue, 3-valued versions of their semantics were proposed by Corman *et al.* and Andreşel *et al.* , allowing to leave some shape assignments as "undetermined".

Next we show the issue of unfounded inferences, which applies to both 2- and 3-valued supported model semantics.

Example 4. *In the KG (B), a total shape assignment as required in Example 3 is possible: simply assign StudShape to* Cameron*, and assign both ProfShape and EmplShape to* Drew*. This situation is questionable because (B) states that* Drew *supervises themselves, which in turn causes an unfounded inference:* Drew *is an employee because* Drew *is an employee. Thus (B) validates the target specification $\mathcal{T} = \{(\text{Drew}, \mathit{ProfShape})\}$ despite the absence of evidence that* Drew *is an employee in the organization.*

3 Preliminaries

We define here some preliminary notions. We start with *property paths*, which allow to express complex path navigations in *data graphs*. These in turn are just finite directed labeled graphs, enriched with built-in monadic predicates. Finally, we recall the *well-founded semantics* for propositional logic programs with negation (Van Gelder, Ross, and Schlipf 1991), which provides a basis for our approach.

Let $\mathrm{N_P}$, $\mathrm{N_C}$, $\mathrm{N_Q}$ be countably infinite, disjoint sets of *property names*, *class names*, and *query names*, respectively.

Property Paths. We let $\mathrm{N_P^{\pm}} = \mathrm{N_P} \cup \{p^- \mid p \in \mathrm{N_P}\}$ and call p^- the *inverse property* of $p \in \mathrm{N_P}$. A *property path* is an expression E built using the following grammar:

$$E ::= r \mid E \circ E \mid E \cup E \mid E^*$$

where $r \in \mathrm{N_P^{\pm}}$. I.e. a property path E is a regular expression over the alphabet $\mathrm{N_P^{\pm}}$.

Data Graphs. Let $\mathbf{N}$ denote a countably infinite set of *nodes*. A *data graph* is a pair $\mathcal{G} = (\Delta^{\mathcal{G}}, \cdot^{\mathcal{G}})$, where $\Delta^{\mathcal{G}} \subseteq \mathbf{N}$ is a non-empty finite set and $\cdot^{\mathcal{G}}$ is a function that assigns to every property name $p \in \mathrm{N_P}$ some binary relation $p^{\mathcal{G}} \subseteq \Delta^{\mathcal{G}} \times \Delta^{\mathcal{G}}$ and to every class name $A \in \mathrm{N_P}$ some subset $A^{\mathcal{G}} \subseteq \Delta^{\mathcal{G}}$. The function $\cdot^{\mathcal{G}}$ is extended to all property paths as follows (next, $\circ$ and $*$ applied on binary relations denote composition and transitive closure, respectively):

$$\begin{aligned}(p^-)^{\mathcal{G}} &= \{(o_2, o_1) \mid (o_1, o_2) \in p^{\mathcal{G}}\}\\ (E_1 \circ E_2)^{\mathcal{G}} &= E_1^{\mathcal{G}} \circ E_2^{\mathcal{G}}\\ (E_1 \cup E_2)^{\mathcal{G}} &= (E_1)^{\mathcal{G}} \cup (E_2)^{\mathcal{G}}\\ (E^*)^{\mathcal{G}} &= (E^{\mathcal{G}})^*\end{aligned}$$

For any E and $o \in \Delta^{\mathcal{G}}$, let $E^{\mathcal{G}}(o) = \{o' \mid (o, o') \in E^{\mathcal{G}}\}$.

We also assume the presence of *built-in* monadic predicates (or, monadic *queries*) that select a subset of nodes in a given data graph. For this, we assume that $\cdot^{\mathcal{G}}$ maps each $q \in \mathrm{N_Q}$ to a set $q^{\mathcal{G}} \subseteq \Delta^{\mathcal{G}}$ in some predetermined way.

Well-founded Semantics of Logic Programs. Assume a countably infinite set $\mathrm{N_{pa}}$ of *propositional atoms*, also called *positive literals*. A *negative literal* is an expression of the form $\neg a$, where $a \in \mathrm{N_{pa}}$. A *literal* is either a positive or a negative literal. A *rule* ρ is an expression of the form

$$h \leftarrow L_1, \ldots, L_n \tag{1}$$

where $h \in \mathrm{N_{pa}}$, and $L_1, \ldots, L_n$ are literals. The atom h is called the *head* of ρ, denoted $\mathsf{head}(\rho)$. We let $\mathsf{body}(\rho) = \{L_1, \ldots, L_n\}$. A *program* P is any finite set of rules.

A (3-valued) *interpretation* I is any set of literals such that $\{a, \neg a\} \not\subseteq I$ for any a. Intuitively, if a is a propositional atom such that $a \in I$, then a is *true* in I. If $\neg a \in I$, then a is *false* in I. If neither $a \in I$ nor $\neg a \in I$, then the truth value of a is *undefined* in I.

A set U of propositional atoms is called an *unfounded set* w.r.t. an interpretation I and a program P, if at least one of the following holds for each rule $\rho \in P$ with $\mathsf{head}(\rho) \in U$:

(i) there is an atom $b \in \mathsf{body}(\rho)$ s.t. $b \in U$ or $\neg b \in I$, or

(ii) there is a negative literal $\neg b \in \mathsf{body}(\rho)$ s.t. $b \in I$.

For any program P and interpretation I, there is a unique $\subseteq$-maximal unfounded set w.r.t. P and I, which we denote $U_P(I)$. This claim follows from the easy observation that "unfoundedness" is preserved under union: if U_1, U_2 are unfounded sets w.r.t. P and I, then $U_1 \cup U_2$ is also an unfounded set w.r.t. P and I.

For a program P, we define a pair T_P and W_P of operators that map interpretations to interpretations as follows:

$$T_P(I) = \{\mathsf{head}(\rho) \mid \mathsf{body}(\rho) \subseteq I, \rho \in P\}$$
$$W_P(I) = T_P(I) \cup \{\neg a \mid a \in U_P(I)\}$$

Intuitively, by applying the operator W_P on an interpretation I, we augment I with some new literals: (i) first we include $T_P(I)$ that contains immediate consequences of the rules in P and the literals in I, and then (ii) we add the negation of all atoms that can be safely assumed false according to $U_P(I)$. The operator W_P is monotonic, i.e., $W_P(I_1) \subseteq W_P(I_2)$ holds whenever $I_1 \subseteq I_2$. Thus W_P has the least fixpoint $\mathit{lfp}(W_P)$. In the well-founded semantics for logic programs above, precisely this set $\mathit{lfp}(W_P)$ is the intended meaning of a program P and it is called the *well-founded model of* P. We let $\mathit{WFS}(P) = \mathit{lfp}(W_P)$. In more detail, we can compute $\mathit{WFS}(P)$ by considering the sequence $I_0, I_1, \ldots$ with $I_0 = \emptyset$ and $I_i = W_P(I_{i-1})$ for all $i > 0$. Due to the monotonicity of W_P, this sequence reaches a j such that $I_{j+1} = I_j$, i.e. it reaches a fixpoint. Then $\mathit{WFS}(P) = I_j$.

Example 5. *Assume a program P with the following rules:*

$$a \leftarrow b, c \qquad b \leftarrow a, d \qquad d \leftarrow e \qquad c \leftarrow \neg b \qquad e \leftarrow \neg d$$

Let us compute $\mathit{WFS}(P)$. We start with $I_0 = \emptyset$. We have $T_P(I_0) = \emptyset$ and $U_P(I_0) = \{a, b\}$. Thus $I_1 = W_P(I_0) = \{\neg a, \neg b\}$. To compute I_2, first note that $T_P(I_1) = \{c\}$ and $U_P(I_1) = \{a, b\}$. Then $I_2 = W_P(I_1) = \{c, \neg a, \neg b\}$. Observe that $U_P(I_2) = \{a, b\}$ and $T_P(I_2) = \{c\}$, and thus $I_3 = I_2$. Hence $\mathit{WFS}(P) = \{c, \neg a, \neg b\}$.

4 Well-founded Semantics for SHACL

In this section we present a well-founded semantics for SHACL. This is done by adapting in a suitable way the notion of unfounded sets and fixpoint computations that we saw above in the context of logic programming.

We start with an (abstract) syntax of SHACL. It is largely based on the original approach in (Corman, Reutter, and Savkovic 2018); see also (Pareti, Konstantinidis, and Mogavero 2022; Bogaerts, Jakubowski, and Van den Bussche 2022a). SHACL is a large specification that describes many constructs. Fortunately, we can concentrate on a small core language, since the majority of SHACL constructs are expressible as monadic SPARQL queries. Our formalization exploits this fact and 'hides' those constructs using built-in monadic predicates.

Syntax. Let $\mathrm{N_S}$ be a countably infinite set of *shape names*, disjoint from $\mathrm{N_P}$, $\mathrm{N_C}$, and $\mathrm{N_Q}$. A *basic shape expression (BSE)* φ is an expression built using the following grammar:

$$\varphi ::= s \mid \neg s \mid A \mid q \mid s \vee s' \mid \forall E.s \mid \geq_n E.s$$

where $s, s' \in \mathrm{N_S}$, $A \in \mathrm{N_C}$, $q \in \mathrm{N_Q}$, E is a property path, and $n > 0$ is an integer. A *validation rule* (or, *shape constraint*)[1] ρ is an expression of the form

$$s \leftarrow \varphi_1, \ldots, \varphi_n \tag{2}$$

where $s \in \mathrm{N_S}$ and $\varphi_1, \ldots, \varphi_n$ is a non-empty sequence of BSEs. We let $\mathit{head}(\rho) = s$ and $\mathit{body}(\rho) = \{\varphi_1, \ldots, \varphi_n\}$. Next, when considering a set $\mathcal{C}$ of shape constraints, we always require that (a) $\mathit{head}(\rho_1) = \mathit{head}(\rho_2)$ implies $\rho_1 = \rho_2$ for all $\rho_1, \rho_2 \in \mathcal{C}$, and (b) for every shape name s that appears in $\mathcal{C}$, there is $\rho \in \mathcal{C}$ such that $s = \mathit{head}(\rho)$. The above just says that every shape name that appears in $\mathcal{C}$ has exactly one "definition". Concretely, for such a constraint set $\mathcal{C}$ and a shape name s that appears in $\mathcal{C}$, we let $\mathcal{C}(s) = \mathit{body}(\rho)$, where ρ is the unique constraint in $\mathcal{C}$ with $\mathit{head}(\rho) = s$. A *target* is a pair (ℓ, s), where $\ell \in \mathbf{N} \cup \mathrm{N_Q} \cup \mathrm{N_P^{\pm}}$ and $s \in \mathrm{N_S}$. A *SHACL document* is a pair $(\mathcal{C}, \mathcal{T})$, where $\mathcal{C}$ is a set of shape constraints and $\mathcal{T}$ is a set of targets.

Semantics. We are interested in checking if a data graph $\mathcal{G}$ *conforms* to a SHACL document $(\mathcal{C}, \mathcal{T})$. Intuitively, this requires checking if the nodes of $\mathcal{G}$ that are selected via targets in $\mathcal{T}$ have the proper neighborhood in $\mathcal{G}$. The latter is verified via the validation rules in $\mathcal{C}$. E.g., if $(o, s) \in \mathcal{T}$, then we need to check that s can be derived at o via $C(s)$ and possibly other shape constraints in $\mathcal{C}$. This all will be made more formal using a well-founded semantics we define next. For this, we first need our version of 3-valued interpretations.

A *shape atom* is an expression of the form $s(a)$, where $s \in \mathrm{N_S}$ and $a \in \mathbf{N}$. A *negated shape atom* is an expression $\neg s(a)$, where $s(a)$ is an atom. A *(shape) literal* is a possibly negated shape atom. A *(3-valued) interpretation* (for SHACL constraints) is any set S of shape literals such that there is no $s(a) \in S$ with $\neg s(a) \in S$. Intuitively, an atom $s(a) \in S$ means that there is a justification that s holds at a, a literal $\neg s(a) \in S$ means that there is no reason for s to hold at a, while $\{s(a), \neg s(a)\} \cap S = \emptyset$ corresponds to the case where the satisfaction of s at a is *undefined*. For a set K of shape atoms, let $\neg.K = \{\neg s(a) \mid s(a) \in K\}$.

For a given graph $\mathcal{G}$ and an interpretation S, we define two functions $\lfloor\cdot\rceil_S^{\mathcal{G}}$ and $\lceil\cdot\rceil_S^{\mathcal{G}}$ that map every BSE φ to a set of nodes in $\mathcal{G}$. These functions are then extended to handle bodies of constraints (i.e. sets of BSEs) in a natural way. The functions are presented in Figure 2.

[1] We use both terms here, since the latter is used in the Semantic Web yet the former is closer in spirit to logic programming.

$$\lfloor s \rfloor_S^{\mathcal{G}} = \{a \mid s(a) \in S\} \qquad \lceil s \rceil_S^{\mathcal{G}} = \{a \in \Delta^{\mathcal{G}} \mid \neg s(a) \notin S\}$$

$$\lfloor \neg s \rfloor_S^{\mathcal{G}} = \{a \mid \neg s(a) \in S\} \qquad \lceil \neg s \rceil_S^{\mathcal{G}} = \{a \in \Delta^{\mathcal{G}} \mid s(a) \notin S\}$$

$$[q]_S^{\mathcal{G}} = q^{\mathcal{G}} \qquad [A]_S^{\mathcal{G}} = A^{\mathcal{G}} \qquad [s \vee s']_S^{\mathcal{G}} = [s]_S^{\mathcal{G}} \cup [s']_S^{\mathcal{G}} \qquad [\cdot] \in \{\lfloor \cdot \rfloor, \lceil \cdot \rceil\}$$

$$[\forall E.s]_S^{\mathcal{G}} = \{a \in \Delta^{\mathcal{G}} \mid \forall a' \in \Delta^{\mathcal{G}} : (a, a') \in E^{\mathcal{G}} \text{ implies } a' \in [s]_S^{\mathcal{G}}\} \qquad [\cdot] \in \{\lfloor \cdot \rfloor, \lceil \cdot \rceil\}$$

$$[\geq_n E.s]_S^{\mathcal{G}} = \{a \in \Delta^{\mathcal{G}} \mid \#\{a' \mid (a, a') \in E^{\mathcal{G}} \text{ and } a' \in [s]_S^{\mathcal{G}}\} \geq n\} \qquad [\cdot] \in \{\lfloor \cdot \rfloor, \lceil \cdot \rceil\}$$

$$[\varphi_1, \ldots, \varphi_n]_S^{\mathcal{G}} = [\varphi_1]_S^{\mathcal{G}} \cap \ldots \cap [\varphi_n]_S^{\mathcal{G}} \qquad [\cdot] \in \{\lfloor \cdot \rfloor, \lceil \cdot \rceil\}$$

Figure 2: Evaluating BSEs and constraint bodies: upper and lower bounds. Here $\#X$ denotes the cardinality of a set X.

Assume a data graph $\mathcal{G}$ and a constraint $s \leftarrow \varphi$. Suppose we "believe" a set S of shape literals, i.e., we assume that all literals in S are true. Then $\lfloor \varphi \rfloor_S^{\mathcal{G}}$ and $\lceil \varphi \rceil_S^{\mathcal{G}}$ return us the nodes of $\mathcal{G}$ where φ *is certainly true* and where φ *is possibly true*, respectively. Thus $\lfloor \varphi \rfloor_S^{\mathcal{G}}$ can be used to infer positive shape literals: if $a \in \lfloor \varphi \rfloor_S^{\mathcal{G}}$, then we can infer $s(a)$. We can use $\lceil \varphi \rceil_S^{\mathcal{G}}$ to infer negative information: if $a \notin \lceil \varphi \rceil_S^{\mathcal{G}}$, then we can infer $\neg s(a)$. These inferences are formalized next.

Definition 1. *Assume a data graph $\mathcal{G}$ and a set $\mathcal{C}$ of constraints. We define an operator $T_{\mathcal{G},\mathcal{C}}(\cdot)$ that maps interpretations into interpretations as follows:*

$$T_{\mathcal{G},\mathcal{C}}(S) = \{s(a) \mid s \leftarrow \varphi_1, \ldots, \varphi_n \in C, a \in \lfloor \varphi_1, \ldots, \varphi_n \rfloor_S^{\mathcal{G}}\}$$

We are now ready to define the notion of an unfounded set of shape atoms.

Definition 2 (Unfounded set)**.** *Assume an interpretation S, a data graph $\mathcal{G}$, and a set $\mathcal{C}$ of constraints. A set U of shape atoms is called an* unfounded set *w.r.t. S, $\mathcal{G}$ and $\mathcal{C}$, if $a \notin \lceil \mathcal{C}(s) \rceil_{S \cup \neg.U}^{\mathcal{G}}$ for all $s(a) \in U$.*

Assume a graph $\mathcal{G}$, a set U of shape atoms, and assume the shape literals in a set S are true. Intuitively, the atoms in U form an unfounded set (and can thus be simultaneously set to *false*) if none of the shape atoms $s(a) \in U$ can possibly be implied by the associated constraint, assuming the negation of the atoms in U holds (in addition to S being true).

The following property follows from the fact that $U_1 \cup U_2$ is an unfounded set w.r.t. S, $\mathcal{G}$ and $\mathcal{C}$ whenever U_1, U_2 are two unfounded sets w.r.t. S, $\mathcal{G}$ and $\mathcal{C}$.

Proposition 1. *Assume an interpretation S, a data graph $\mathcal{G}$, and a constraint set $\mathcal{C}$. There exists a unique set U such that:*

- *U is an unfounded set w.r.t. S, $\mathcal{G}$ and $\mathcal{C}$, and*
- *there is no $U' \supset U$ that is unfounded set w.r.t. S, $\mathcal{G}$ and $\mathcal{C}$.*

The unique set U in the proposition above is called the *greatest unfounded set* w.r.t. S, $\mathcal{G}$ and $\mathcal{C}$. Assume a data graph $\mathcal{G}$ and a set $\mathcal{C}$ of constraints. We let $U_{\mathcal{G},\mathcal{C}}$ be the operator that maps every interpretation S to the greatest unfounded set w.r.t. S, $\mathcal{G}$ and $\mathcal{C}$. Thus, $U_{\mathcal{G},\mathcal{C}}(S)$ is the maximal set of shape atoms that we can safely set to *false* if we assume that the literals in S are true. We can now finally define a well-founded semantics for SHACL. We define an operator $W_{\mathcal{G},\mathcal{C}}$ that maps interpretations into interpretations as follows: It combines the positive consequences based on the $T_{\mathcal{G},\mathcal{C}}$ operator and the negated atoms of the greatest unfounded set produced by the $U_{\mathcal{G},\mathcal{C}}$ operator. More formally, we define:

$$W_{\mathcal{G},\mathcal{C}}(S) = T_{\mathcal{G},\mathcal{C}}(S) \cup \neg.U_{\mathcal{G},\mathcal{C}}(S)$$

The above operator is monotone, i.e., $W_{\mathcal{G},\mathcal{C}}(S_1) \subseteq W_{\mathcal{G},\mathcal{C}}(S_2)$ whenever S_1, S_2 are two interpretations with $S_1 \subseteq S_2$. Thus $W_{\mathcal{G},\mathcal{C}}$ has the least fixpoint, i.e., there exists S such that (i) $W_{\mathcal{G},\mathcal{C}}(S) = S$, and (ii) there is no $S' \subset S$ with $W_{\mathcal{G},\mathcal{C}}(S') = S'$. We use $\mathit{WFS}(\mathcal{G},\mathcal{C})$ to denote the least fixpoint of $W_{\mathcal{G},\mathcal{C}}$, and call it the *well-founded model* of $\mathcal{G}$ and $\mathcal{C}$. $\mathit{WFS}(\mathcal{G},\mathcal{C})$ can be obtained by constructing a sequence $S_0, S_1, S_2, \ldots$ such that $S_0 = \emptyset$ and $S_{i+1} = W_{\mathcal{G},\mathcal{C}}(S_i)$ until eventually an index j is reached were $S_j = S_{j+1}$. Then $\mathit{WFS}(\mathcal{G},\mathcal{C}) = S_j$.

We can now define validation as follows.

Definition 3. *We say a data graph $\mathcal{G}$ is* valid *against a SHACL document $(\mathcal{C}, \mathcal{T})$, if for all $(\ell, s) \in \mathcal{T}$ we have:*

- *if $\ell \in \mathbf{N}$, then $s(\ell) \in \mathit{WFS}(\mathcal{G},\mathcal{C})$;*
- *if $\ell \in \mathrm{N_C} \cup \mathrm{N_Q}$, then $s(a) \in \mathit{WFS}(\mathcal{G},\mathcal{C})$ for all $a \in \ell^{\mathcal{G}}$;*
- *if $\ell \in \mathrm{N_P^{\pm}}$, then $s(a) \in \mathit{WFS}(\mathcal{G},\mathcal{C})$ for all a s.t. $(a, b) \in \ell^{\mathcal{G}}$.*

Example 6. *Take the graph $\mathcal{G}$ corresponding to (A) in Fig. 1 and a constraint set $\mathcal{C}$ with the three constraints of Fig. 1. One can see that $\mathit{WFS}(\mathcal{G},\mathcal{C})$ includes $\mathit{ProfShape}(\mathrm{Blake})$, $\mathit{EmplShape}(\mathrm{Blake})$, $\mathit{EmplShape}(\mathrm{Alex})$, $\mathit{StudShape}(\mathrm{Cameron})$. Observe that $\mathit{ProfShape}(\mathrm{Alex}) \notin \mathit{WFS}(\mathcal{G},\mathcal{C})$ and $\neg \mathit{ProfShape}(\mathrm{Alex}) \notin \mathit{WFS}(\mathcal{G},\mathcal{C})$, i.e., membership of* Alex *in ProfShape is undetermined, which is intuitive as discussed in Example 1. For the remaining shape atoms over the signature of $\mathcal{G}$ and $\mathcal{C}$, we have that $\mathit{WFS}(\mathcal{G},\mathcal{C})$ contains their negation. Thus $\mathcal{G}$ is valid against $(\mathcal{C}, \{(\mathrm{Blake}, \mathit{ProfShape})\})$ but not valid w.r.t. $(\mathcal{C}, \{(\mathrm{Drew}, \mathit{ProfShape})\})$.*

We remark here that performing validation as described in Definition 3 is computationally tractable. Indeed, checking whether $\mathcal{G}$ is valid against a SHACL document $(\mathcal{C}, \mathcal{T})$ is a PTIME-complete problem. The argument for the upper bound is essentially the same as for the complexity of computing the well-founded model of a propositional logic program (Van Gelder, Ross, and Schlipf 1991). Indeed, there are only polynomially many shape literals over the signature

of $\mathcal{C}$ and the nodes in $\mathcal{G}$. Applying the operator $W_{\mathcal{G},\mathcal{C}}$ at any stage towards reaching the least fix-point needs only polynomial time in the size of $\mathcal{G}$ and $\mathcal{C}$. For this, the main aspect to observe is that computing $\lfloor\varphi\rfloor_S^{\mathcal{G}}$ and $\lceil\varphi\rceil_S^{\mathcal{G}}$ for a given $\varphi, \mathcal{G}, S$ is feasible in polynomial time. The lower bound follows directly from the fact that our validation problem subsumes the entailment problem for propositional Horn programs.

5 Connecting to Logic Programming

Our goal next is to connect the well-founded semantics for SHACL introduced above to classic the well-founded semantics of logic programs with default negation. This has two purposes. First, we want to illustrate that the above SHACL semantics is in fact based on a well-understood formalism in logic programming, opening the way to transfer results from the latter area to an area of the Semantic Web that is receiving significant attention nowadays. The second purpose is to pave the way to an implementation of SHACL validation that reuses existing engines for computing the well-founded semantics of logic programs.

From Logic Programs to SHACL Assume a logic program P and an atom a. Suppose we want to check whether $a \in \mathit{WFS}(P)$. We show how this can be done by a reduction to validation in SHACL under the well-founded semantics.

We note first that using fresh propositional atoms, P can be normalized into a program that has the following form:

- If P has more than one rule with an atom b in the head, then P has exactly two rules with b in the head and those rules have the form $b \leftarrow b_1$ and $b \leftarrow b_2$, for some b_1, b_2. We say such b *depends on two rules* in P.
- If b occurs in P, then P has a rule with b in the head.

Note that this normalization can be achieved in polynomial time. To construct a desired set $\mathcal{C}$ of shape constraints, for every propositional atom p that appears in P, take a fresh shape name s_p. Moreover, for every fact $p \leftarrow$ in P, let A_p be a fresh class name. We build $\mathcal{C}$ by adding the following constraints for each atom p that occurs in P, depending on the kind of p:

- The atom p depends on two rules, i.e. some distinct $p \leftarrow p_1$ and $p \leftarrow p_2$ are in P. Then add $s_p \leftarrow s_{p_1} \vee s_{p_2}$ to $\mathcal{C}$.
- A non-fact rule $p \leftarrow b_1, \dots, b_k, \mathit{not}\ b_{k+1} \dots, \mathit{not}\ b_m$ belongs to P and it is the only rule with p in the head. Then add $s_p \leftarrow s_{b_1}, \dots, s_{b_k}, \mathit{not}\ s_{b_{k+1}} \dots, \mathit{not}\ s_{b_m}$ to $\mathcal{C}$.
- The fact $p \leftarrow$ belongs to P. Then add $s_p \leftarrow A_p$ to $\mathcal{C}$.

Since we want to check $a \in \mathit{WFS}(P)$, we set the target as $\mathcal{T} = \{(o, s_a)\}$. We define a data graph $\mathcal{G}$ that encodes the facts of P as follows. We set $\Delta^{\mathcal{G}} = \{o\}$ and let $A_p^{\mathcal{G}} = \{o\}$ for all facts $p \leftarrow$ of P.

This completes the construction of $\mathcal{C}, \mathcal{T}, \mathcal{G}$ from the program P. The following can be seen from the definitions of the well-founded semantics for SHACL and propositional logic programs:

Proposition 2. *Assume $\mathcal{C}, \mathcal{T}, \mathcal{G}$ constructed from P as shown above. For a SHACL interpretation S over the signature of $\mathcal{C}$, let $pa(S) = \{p \mid s_p(o) \in S\} \cup \{\neg p \mid \neg s_p(o) \in S\}$. Then $\mathit{WFS}(P) = pa(\mathit{WFS}(\mathcal{G}, \mathcal{C}))$. Thus $a \in \mathit{WFS}(P)$ iff $\mathcal{G}$ is valid against $(\mathcal{C}, \mathcal{T})$.*

Proof (Sketch). Take the sequence $I_0 \subseteq I_1 \subseteq \dots$ with $I_0 = \emptyset$ and $I_{i+1} = W_P(S_i)$. Recall that $\mathit{WFS}(I) = I_j$ for the smallest j with $I_j = I_{j+1}$. Further, take the sequence $S_0 \subseteq S_1 \subseteq \dots$ with $S_0 = \emptyset$ and $S_{i+1} = W_{\mathcal{G},\mathcal{C}}(S_i)$. Similar to above, $\mathit{WFS}(\mathcal{G}, \mathcal{C}) = S_k$ for the smallest k with $S_k = S_{k+1}$. For the claim, it suffices to show that $I_\ell = pa(S_\ell)$ for all $\ell \geq 0$, which we can do by induction. For $\ell = 0$, $I_0 = pa(S_0)$ since $I_0 = S_0 = \emptyset$. For the inductive case, assume $I_\ell = pa(S_\ell)$ for some $\ell \geq 0$. Then the following can be verified. First, $T_P(I_\ell) = pa(T_{\mathcal{G},\mathcal{C}}(S_\ell))$. Second, $U_P(I_\ell) = pa(U_{\mathcal{G},\mathcal{C}}(S_\ell))$. From this, it follows that $W_P(I_\ell) = pa(W_{\mathcal{G},\mathcal{C}}(S_\ell))$, and the claim follows since $W_P(I_\ell) = I_{\ell+1}$ and $W_{\mathcal{G},\mathcal{C}}(S_\ell) = S_{\ell+1}$. □

From SHACL to Logic Programs. Next we turn to a translation from SHACL to logic programs, which is the first step towards an implementation of SHACL discussed in the next section. Specifically, for a given data graph $\mathcal{G}$ and a constraint set $\mathcal{C}$ we construct a logic program P such that $\mathit{WFS}(\mathcal{G}, \mathcal{C})$ corresponds to $\mathit{WFS}(P)$. In this way, validation over $\mathcal{G}$ reduces to query answering over P.

Assume a data graph $\mathcal{G}$ and a constraint set $\mathcal{C}$. To construct P, we use propositional atoms of the form u_o^s and u_o^φ, where s and φ are a shape name and a basic shape expression that appear in $\mathcal{C}$, respectively, and $o \in \Delta^{\mathcal{G}}$. We start the construction of P by adding the following rule

$$u_o^s \leftarrow u_o^{\varphi_1}, \dots, u_o^{\varphi_n}$$

for all nodes $o \in \Delta^{\mathcal{G}}$ and all $s \leftarrow \varphi_1, \dots, \varphi_n$ in $\mathcal{C}$.

The rest of P deals with the remaining BSEs that appear in $\mathcal{C}$. For all nodes $o \in \Delta^{\mathcal{G}}$, we add to P the following:

(1) If a query name q appears in $\mathcal{C}$ and $o \in q^{\mathcal{G}}$, we add

$$u_o^q \leftarrow$$

(2) If a class name A appears in $\mathcal{C}$ and $o \in A^{\mathcal{G}}$, we add

$$u_o^A \leftarrow$$

(3) For each $\neg s$ in $\mathcal{C}$, we add

$$u_o^{\neg s} \leftarrow \neg u_o^s$$

(4) For each $s_1 \vee s_2$ in $\mathcal{C}$, we add

$$u_o^{s_1 \vee s_2} \leftarrow u_o^{s_1}$$

$$u_o^{s_1 \vee s_2} \leftarrow u_o^{s_2}$$

(5) For each $\forall E.s$ in $\mathcal{C}$, we add

$$u_o^{\forall E.s} \leftarrow u_{o_1}^s, \dots, u_{o_n}^s$$

where $\{o_1, \dots, o_n\} = E^{\mathcal{G}}(o)$

(6) For each $\geq_n E.s$ in $\mathcal{C}$, we add

$$u_o^{\geq_n E.s} \leftarrow u_{o_1}^s, \dots, u_{o_n}^s$$

for all sets $\{o_1, \dots, o_n\} \subseteq E^{\mathcal{G}}(o)$ of cardinality n.

This completes the definition of P, and we can formulate a correspondence result:

Proposition 3. *Assume a program P built from a graph $\mathcal{G}$ and a constraint set $\mathcal{C}$ as shown above. For a set Q of literals in the signature of P, let $sa(Q) = \{s(o) \mid u_o^s \in Q\} \cup \{\neg s(o) \mid \neg u_o^s \in Q\}$. Then following holds:*

$$WFS(\mathcal{G},\mathcal{C}) = sa(WFS(P))$$

Proof (Sketch). For this claim, we slightly modify the definition of the well-founded semantics of logic programs: we define the operator W_P as $W_P(I) = T_P^*(I) \cup \{\neg a \mid a \in U_P(I)\}$, where $T_P^*(I)$ denotes the interpretation J that is obtained by applying T_P on I exhaustively until a fixpoint is reached, i.e. formally, $T_P^*(I)$ is the least fixpoint of the monotone operator $T_{P,I}(K) = I \cup T_P(K)$. Intuitively, at one iteration of applying W_P we collect not only the *immediate* positive consequences of rules in P but do this exhaustively. One check that this change is harmless; it preserves the original well-founded semantics of logic programs. We perform a similar change to the definition of the well-founded semantics of SHACL: we define $W_{\mathcal{G},\mathcal{C}}$ as $W_{\mathcal{G},\mathcal{C}}(S) = T_{\mathcal{G},\mathcal{C}}^*(S) \cup \neg.U_{\mathcal{G},\mathcal{C}}(S)$, where $T_{\mathcal{G},\mathcal{C}}^*(S)$ denotes the interpretation S' that is obtained by applying $T_{\mathcal{G},\mathcal{C}}$ on S exhaustively until a fixpoint is reached. Again, one can check that this modification does not change the semantics of SHACL in this paper. Due the way we redefined the operators W_P and $W_{\mathcal{G},\mathcal{C}}$, one can verify that $(*)$ $sa(W_P(I)) = W_{\mathcal{G},\mathcal{C}}(sa(I))$ holds for any consistent set I of literals over the signature of P.

Take the sequence $I_0 \subseteq I_1 \subseteq \dots$ with $I_0 = \emptyset$ and $I_{i+1} = W_P(S_i)$. Further, take the sequence $S_0 \subseteq S_1 \subseteq \dots$ with $S_0 = \emptyset$ and $S_{i+1} = W_{\mathcal{G},\mathcal{C}}(S_i)$. For the claim, it suffices to show by induction that $S_\ell = sa(I_\ell)$ for all $\ell \geq 0$. The base case is trivial: $S_0 = sa(I_0)$ since $S_0 = I_0 = \emptyset$. Assume $S_\ell = sa(I_\ell)$ holds for some $\ell \geq 0$. By exploiting $(*)$, we get that $S_{\ell+1} = W_{\mathcal{G},\mathcal{C}}(S_\ell) = W_{\mathcal{G},\mathcal{C}}(sa(I_\ell)) = sa(W_P(I_\ell)) = sa(I_{\ell+1})$, as desired. ☐

As a consequence of the above, given any target $\mathcal{T}$, we can check whether $\mathcal{G}$ validates against $(\mathcal{C},\mathcal{T})$ by first building P as above and then inspecting $WFS(P)$. E.g., if $\mathcal{T}$ contains some (o, s), then we must verify that $u_o^s \in WFS(P)$; this extends naturally to other kinds of statements in $\mathcal{T}$.

The above reduction from SHACL validation to reasoning in a logic program provides a foundation but is not directly suitable for implementation. E.g., due to a possible very large size of $\mathcal{G}$ in practice, the program P can easily become too large for existing deductive database engines. Furthermore, BSEs of the form $\geq_n E.s$ cause an exponential explosion in the value n (see point (6) above where we traverse subsets of $E^{\mathcal{G}}(o)$ of size n). Finally, the construction of P above does not take into account validation targets, i.e., P can be used for validating $\mathcal{G}$ against $(\mathcal{C},\mathcal{T})$ for any $\mathcal{T}$. However, if we are only interested in validating using a specific $\mathcal{T}$, then P may contain a large amount of rules are irrelevant. For this reason, our implementation employs a powerful but simple method to compute a smaller module of P that is sufficient to reason about $\mathcal{T}$. We formulate this optimization for general logic programs under the well-founded semantics. We note that this is closely related to the notion of *splitting sets* in logic programs under the stable model semantics (Lifschitz and Turner 1994).

Definition 4. *Let P be a program and A a set of atoms. Let $mod(P, A)$ be the smallest set such that:*

- *$mod(P, A)$ contains every rule from P whose head atom belongs to A.*
- *if $h \leftarrow L_1, \dots, L_n \in mod(P, A)$, then $mod(P, A)$ contains all rules of P whose head atom appears in some $L_1, \dots, L_n$.*

Proposition 4. *Let P be a program and A a set of atoms. We have that $A \subseteq WFS(P)$ iff $A \subseteq WFS(mod(P, A))$.*

Proof (Sketch). Let $P_0 = mod(P, A)$ and $P_1 = P \setminus P_0$. The proposition follows directly from the equality

$$WFS(P_0) = WFS(P) \cap \{a, \neg a \mid a \text{ appears in } P_0\}.$$

Indeed, if $A \subseteq WFS(P)$, then for each $a \in A$ there is a rule $\rho \in P$ with a in its head. By definition of $mod(P, A)$, we have $\rho \in P_0$ and thus that a appears in P_0. Then due the above equality we get $a \in WFS(P_0)$. If $A \subseteq WFS(P_0)$, then $A \subseteq WFS(P)$ trivially holds since $WFS(P_0) \subseteq WFS(P)$ due to the above equality.

To see the equality, observe that the rules in P_1 are such that their head atoms do not appear in P_0. Intuitively, this means that we can obtain $WFS(P)$ by first computing $WFS(P_0)$ and then evaluating P_1 on top of $WFS(P_0)$ without deriving new positive or negative literals over the atoms that appear in P_0. I.e., $WFS(P_0)$ can be expanded to $WFS(P)$ without adding literals over the atom signature of P_0. Technically, consider the sequence $H_0, H_1, \dots$ where $H_0 = WFS(P_0)$ and $H_i = W_{P_1}(H_{i-1})$ for all $i > 0$. Due to monotonicity of W_{P_1}, there is j such that $H_{i+1} = H_i$. Note that in this sequence no atoms over the signature of P_0 are added. Using the basic properties of the consequence operators, it can be verified that $H_i = WFS(P)$. ☐

6 Practical Validation of SHACL under WFS

In this section we present a formal algorithm to extract a ground logic program from a given data graph to see if it is valid under a given SHACL document. This logic program will correspond exactly with the one given in Section 5.

In Section 5, the function mod is introduced, to produce a subset of a program that is sufficient to derive a specified set of atoms. For the purpose of validating a data graph $\mathcal{G}$ under a SHACL document $(\mathcal{C},\mathcal{T})$, these are exactly those atoms u_a^s which "correspond" to some target $(\ell, s) \in \mathcal{T}$ where $a \in \ell^{\mathcal{G}}$ (or resp. $(a, b) \in \ell^{\mathcal{G}}$ if $\ell \in \mathrm{N}_{\mathrm{P}}^{\pm}$). In our implementation, we realize this behavior of producing such a sufficient subset via the notion of relevant targets.

Definition 5 (Relevant Targets of a Shape)**.** *Given a data graph $\mathcal{G}$, a SHACL document $(\mathcal{C},\mathcal{T})$, and a given shape constraint s, the set of* relevant targets *of s, written as $\mathtt{RT}(s)$, is*

Algorithm 1: Extract Logic Program

Input: $\mathcal{G}$: a data graph, $(\mathcal{C}, \mathcal{T})$: a SHACL document
Output: P: a logic program

1 **begin**
2 $\quad P := \emptyset$
3 $\quad$ **foreach** $c = (s \leftarrow \varphi_1, \ldots, \varphi_n) \in \mathcal{C}$ **do**
4 $\quad\quad possibleTargets := \mathtt{RT}(s)$
5 $\quad\quad$ **foreach** $q \in body(c)$ **do**
6 $\quad\quad\quad possibleTargets := q^{\mathcal{G}} \cap possibleTargets$
7 $\quad\quad$ **foreach** $a \in possibleTargets$ **do**
8 $\quad\quad\quad P := P \cup (u_a^s \leftarrow \{u_a^{\varphi_i} \mid \varphi_i \neq q\})$
9 $\quad\quad\quad$ **foreach** $\varphi_i \in \{\varphi_1, \ldots, \varphi_n\}$ **do**
10 $\quad\quad\quad\quad$ **if** $\varphi_i = \forall E.s'$ **then**
11 $\quad\quad\quad\quad\quad P := P \cup \{u_a^{\forall E.s'} \leftarrow \{u_b^{s'} \mid b \in E^{\mathcal{G}}(a)\}\}$
12 $\quad\quad\quad\quad$ **else if** $\varphi_i = \geq_n E.s'$ **then**
13 $\quad\quad\quad\quad\quad P := P \cup \bigcup\{(u_a^{\geq_n E.s'} \leftarrow \{u_b^{s'} \mid b \in U\}) \mid U \in \binom{E^{\mathcal{G}}(a)}{n}\}$
14 $\quad\quad\quad\quad$ **else if** $\varphi_i = \neg s'$ **then**
15 $\quad\quad\quad\quad\quad P := P \cup \{u_a^{\neg s'} \leftarrow \neg u_a^{s'}\}$
16 $\quad\quad\quad\quad$ **else if** $\varphi_i = s' \vee s''$ **then**
17 $\quad\quad\quad\quad\quad P := P \cup \{u_a^{s' \vee s''} \leftarrow u_a^{s'}\} \cup \{u_a^{s' \vee s''} \leftarrow u_a^{s''}\}$
18 $\quad$ **return** P

defined as follows:

$$\begin{aligned}\mathtt{RT}(s) = &\{a \in \mathbf{N} \mid (a, s) \in \mathcal{T}\} \cup \{a \mid (\phi, s) \in \mathcal{T}, a \in \phi^{\mathcal{G}}\}\\ &\cup \{b \mid (\forall E.s) \in \mathcal{C}(s'), a \in \mathtt{RT}(s'), (a, b) \in E^{\mathcal{G}}\}\\ &\cup \{b \mid (\geq_n E.s) \in \mathcal{C}(s'), a \in \mathtt{RT}(s'), (a, b) \in E^{\mathcal{G}}\}\\ &\cup \{a \mid (\neg s) \in \mathcal{C}(s'), a \in \mathtt{RT}(s')\}\\ &\cup \{a \mid (s \vee s'') \in \mathcal{C}(s'), a \in \mathtt{RT}(s')\}\end{aligned}$$

where ϕ is either a class, a query or a property.

Description of Algorithm 1. In Algorithm 1 we can see the procedure to extract a logic program from a given SHACL document $(\mathcal{C}, \mathcal{T})$ and data graph $\mathcal{G}$. The program iterates over all shape constraints in $\mathcal{C}$ in a loop running between lines 3 and 17. We first collect all the relevant targets of a shape at line 4. Next, in the loop running between lines 5 and 6 we filter out any nodes from this set that violate any BSEs which are of the form q. The algorithm continues in a loop over all remaining nodes. For each such node a, we construct a logic program rule, seen in line 8. Next, we iterate over all BSEs in the constraints in a loop running between lines 9 and 17. We make a case distinction on φ_i. For all four of these cases, seen on lines 11, 13, 15 and 17, we follow the same construction as seen in Section 5. The final program, consisting of all such rules for all shape constraints, is returned at line 18.

On the use of SPARQL. In Algorithm 1 we use the evaluation function $\cdot^{\mathcal{G}}$ whenever we need to evaluate something over the data graph $\mathcal{G}$. We understand this to be implemented via suitable SPARQL queries. In our implementation presented later, we will also not issue multiple queries per shape, but instead collect all the necessary checks against the data graph into *a single query per shape constraint*.

7 Implementation and Experiments

We present here our implementation of a prototype SHACL validator, called ShaWell and proceed to show its practical applicability in solving recursive SHACL via comparison with other state-of-the-art validators. The source code is publicly available[2], where there are also instructions to easily build and install it on different operating systems.

ShaWell. ShaWell is written in the programming language Go and implements Algorithm 1. It incorporates two key optimizations, however. The first is that instead of querying the data graph multiple times, as a naive reading of Algorithm 1 would suggest, ShaWell issues exactly one SPARQL query per shape. The second optimization is that in the produced logic program, counting is encoded by a polynomial number of rules, as opposed to exponentially many rules that Algorithm 1 would produce. In case of recursion, ShaWell produces a logic program and uses DLV (Leone et al. 2006) to produce a well-founded model. For non-recursive SHACL documents, ShaWell eschews the need for a logic program solver, and directly computes whether the data graph is valid under the given SHACL document.

To compare the effectiveness and performance of ShaWell, we decided to compare it with other SHACL validators that provide support for recursive SHACL documents. For these experiments, we focus on three other systems that support SHACL documents with recursive shapes: Shacl-Sparql[3] from Corman et al. (Corman et al. 2019), SHaclEX[4] developed by Jose Emilio Labra Gayo (Labra Gayo 2016) and the SHACL validator that is a part of the Apache Jena library[5].

Methodology. We use two sets of recursive SHACL documents for our experiments. The first is a test set that was used in (Corman et al. 2019), which expects (parts of) DBPedia as the data graph, and uses custom shape constraints. The second set is from (Ahmetaj et al. 2022b), which also uses DBPedia for the data graph and provides two constraint sets, C1 and C2, and 20 different target expressions, leading to an overall 40 unique SHACL documents. The triple store GraphDB at version 10.4.1 was used as the SPARQL endpoint. Our dataset is a complete copy of DBPedia, as of the date of our experiment[6], with over 120 million triples. Our test machine has an Intel Xeon E5-2620 v2 CPU, clocked at 2.10GHz and with 64 GB of RAM and a 120GB SSD. It is running Ubuntu 22.04.3, under the kernel version 5.15.0-89.

We can see in Table 1 the results of the validation process for all three systems. In the table, the first two columns state the test set and its size understood as the number of tests in the set. Next, are four groups of four columns each, for each of the four systems that are being compared. Each group consists of the column (TO), indicating the number of tests for which the system took more than 15 minutes to validate, followed by the column (avg) (resp. (med) and (stdev))

[2] https://github.com/cem-okulmus/shawell
[3] https://github.com/rdfshapes/shacl-sparql
[4] https://www.weso.es/shaclex
[5] https://jena.apache.org/download/index.cgi
[6] From December 2022 (updated annually). Can be found at http://dev.dbpedia.org/Download_DBpedia.

Test Set	Size	ShaWell				SHaclEX (Labra Gayo 2016)				Shacl-Sparql (Corman et al. 2019)				Apache Jena			
		TOs	avg	med	stdev	TOs	avg	med	stdev	TOs	avg	med	stdev	TOs	avg	med	stdev
C1	20	1	113.37	1.48	199.66	7	31.59	3.98	67.92	Fails to parse				20	-	-	-
C2	20	3	106.23	1.42	212.19	7	31.62	3.96	65.33	Fails to parse				20	-	-	-
rec	3	0	181.27	195.15	80.72	3	-	-	-	0	0.25	0.25	0.00	3	-	-	-

Table 1: Results of our experiments, comparing ShaWell, SHaclEX, Shacl-Sparql, and Apache Jena. The running times reported here are given in seconds and rounded to 2 decimal places. Timeout was set to 15 minutes.

Test Set	ShaWell				
	Total	SPARQL	LP	DLV	Parse
C1	113.37	85.14	4.08	16.74	7.26
C2	106.23	80.08	4.34	17.58	4.11
rec	181.27	133.31	1.58	44.16	2.09

Table 2: This table shows a more detailed breakdown of the running time of ShaWell. All aggregations are over values in seconds.

which indicates the average (resp. median and standard deviation of) running time for all tests which did not time out for the system in question. Note that in Table 1 some entries are left blank, indicated by the symbol (-). This indicates that all tests of a given test set timed out, and thus there is no value that can be produced for the last three columns of the group. In addition to this, another type of failure can be seen for Shacl-Sparql, which failed to parse any tests from test sets C1 and C2. The type of error produced indicates that the developers do not support – as of the time of our experiments – the entirety of the SHACL core standard.

Discussion of Experiments. We can see in Table 1 that ShaWell produces the least amount of timeouts for test sets C1 and C2, and matches Shacl-Sparql in producing no timeouts for test set rec. In terms of average running times, it is slower than SHaclEX and significantly slower than Shacl-Sparql (for the test set rec that Shacl-Sparql can actually parse and validate). Combined with the lower timeout value, this may indicate that there is potential to further improve the performance of our implementation.

To get further insight into the running time of ShaWell, we show in Table 2 a detailed breakdown of its running times. For each of the three tests sets, we show the average running time as "Total", and this is followed by how much time on average is spent on various phases of the algorithm. The first phase, "SPARQL", indicates how much time was spent on average on the execution of SPARQL queries. The column "LP" represent how much time was spent producing the ground logic program. The column "DLV" then indicates how much time the computation of the well-founded model in the solver DLV took on average. The last column "Parse" shows the time spent parsing the output of DLV.

We can see that for C1 the SPARQL queries took 85.14 seconds out of 113.37, so around 75% of the total. For C2 we also get around 75% and for "rec" it is a slightly lower 73% that is spent on SPARQL queries. As such, the largest performance gains would likely come from optimizing the SPARQL queries, as well as potential optimizations on the query engine. The actual amount of time time spent answering the logic program seems comparatively low.

8 Discussion and Conclusion

In this paper we have defined a well-founded semantics for SHACL constraints with cyclic dependencies. This semantics is not only intuitive and based on well-established principles in logic programming and deductive databases, but it also has advantages over the previous approaches (based on supported and stable models) in terms of semantic and computational properties, i.e., it avoids unfounded justifications and is tractable. We presented our prototype implementation of the transformation from SHACL into logic programs via SPARQL queries, which makes use of a deductive database engine to compute the well-founded model of the resulting program. Our validator called ShaWell is publicly available and serves to make the well-founded semantics for recursive SHACL practically available to a larger audience.

We recall that (Andreşel et al. 2020) draws a connection between the usual well-founded semantics for logic programs with default negation and the problem of *cautious validation* under the 3-valued stable model semantics for SHACL, which is due to the classic result by Przymusinski (Przymusinski 1990). Thus a definition of a well-founded semantics for SHACL can be derived from (Andreşel et al. 2020), but it is not explicit in this previous work. The authors of (Bogaerts and Jakubowski 2021) study the semantics of recursive SHACL in the context of *Approximation Fixpoint Theory (AFT)*, which is an abstract framework that captures most standard semantics of formalisms for nonmonotonic reasoning, including the supported model, the stable model, and the well-founded semantics of logic programs. When applied to SHACL, AFT allows to induce these three and other semantics for SHACL. However, AFT is meant to be a unifying framework for capturing multiple semantics, and hence it provides limited insights into the specific semantics in terms of complexity, knowledge representation, and adequacy for SHACL.

There are many research questions for future work. For instance, the computational complexity of satisfiability and implication of SHACL constraints under the well-founded semantics is a natural next topic for investigations. Some initial results in this area for the semantics introduced previously were presented in (Leinberger et al. 2020; Pareti, Konstantinidis, and Mogavero 2022). Another direction is to study *explanations* and *repairs* of violations of SHACL constraints under the well-founded semantics: these topics are natural as the SHACL standard calls for (but does not specify the details of) the so-called *validation reports* to support users and applications. Some initial work on explaining and repairing violations of SHACL constraints was reported in (Ahmetaj et al. 2021; Ahmetaj et al. 2022a).

Acknowledgments

We thank Andrian Chmurovič who, during a student project at TU Wien, worked on an separate implementation of a SHACL validator and provided useful comments on the preliminary version of this paper. This work was partially supported by the Austrian Science Fund (FWF) projects P30360 and P30873, and by the Wallenberg AI, Autonomous Systems, and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation.

References

Ahmetaj, S.; David, R.; Ortiz, M.; Polleres, A.; Shehu, B.; and Šimkus, M. 2021. Reasoning about Explanations for Non-validation in SHACL. In *Proc. of KR 2021*, 12–21.

Ahmetaj, S.; David, R.; Polleres, A.; and Šimkus, M. 2022a. Repairing SHACL constraint violations using answer set programming. In *Proc. of ISWC 2022*, volume 13489 of *Lecture Notes in Computer Science*, 375–391. Springer.

Ahmetaj, S.; Löhnert, B.; Ortiz, M.; and Šimkus, M. 2022b. Magic shapes for SHACL validation. *Proc. VLDB Endow.* 15(10):2284–2296.

Andreşel, M.; Corman, J.; Ortiz, M.; Reutter, J. L.; Savkovic, O.; and Šimkus, M. 2020. Stable model semantics for recursive SHACL. In *WWW '20: The Web Conference 2020*, 1570–1580. ACM / IW3C2.

Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; and Patel-Schneider, P. F., eds. 2003. *The description logic handbook: theory, implementation, and applications*. USA: Cambridge University Press.

Bogaerts, B., and Jakubowski, M. 2021. Fixpoint semantics for recursive SHACL. In *Proc. ICLP 2021 (Technical Communications)*, volume 345 of *EPTCS*, 41–47.

Bogaerts, B.; Jakubowski, M.; and Van den Bussche, J. 2022a. Expressiveness of SHACL features. In Olteanu, D., and Vortmeier, N., eds., *Proc. of ICDT 2022*, volume 220 of *LIPIcs*, 15:1–15:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Bogaerts, B.; Jakubowski, M.; and Van den Bussche, J. 2022b. SHACL: A description logic in disguise. In *Proc. of LPNMR 2022*, volume 13416 of *Lecture Notes in Computer Science*, 75–88. Springer.

Corman, J.; Florenzano, F.; Reutter, J. L.; and Savkovic, O. 2019. Validating SHACL constraints over a SPARQL endpoint. In *Proc. of ISWC 2019*, volume 11778 of *LNCS*, 145–163. Springer.

Corman, J.; Reutter, J. L.; and Savkovic, O. 2018. Semantics and validation of recursive SHACL. In *Proc. of ISWC 2018*, volume 11136 of *LNCS*, 318–336. Springer.

Knublauch, H., and Kontokostas, D. 2017. Shapes constraint language (SHACL). W3C Recommendation. https://www.w3.org/TR/shacl/.

Labra Gayo, J. E. 2016. SHaclEX - SHACL/ShEx implementation. https://github.com/weso/shaclex.

Leinberger, M.; Seifer, P.; Rienstra, T.; Lämmel, R.; and Staab, S. 2020. Deciding SHACL shape containment through description logics reasoning. In *Proc. of ISWC 2020*. Springer.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.* 7(3):499–562.

Lifschitz, V., and Turner, H. 1994. Splitting a logic program. In *Proc.of ICLP 1994*, 23–37.

Pareti, P., and Konstantinidis, G. 2021. A review of SHACL: from data validation to schema reasoning for RDF graphs. In Šimkus, M., and Varzinczak, I., eds., *Tutorial Notes of the International Reasoning Web Summer School 2021 (RW 2021)*, volume 13100 of *LNCS*. Springer.

Pareti, P.; Konstantinidis, G.; and Mogavero, F. 2022. Satisfiability and containment of recursive SHACL. *Journal of Web Semantics* 74:100721.

Przymusinski, T. C. 1990. The well-founded semantics coincides with the three-valued stable semantics. *Fundam. Inform.* 13:445–463.

Van Gelder, A.; Ross, K. A.; and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *J. ACM* 38(3):619–649.