Stable Model Semantics for Recursive SHACL

Anonymous Author(s)*

ABSTRACT

SHACL (Shapes Constraint Language) is a W3C recommendation for validating graph-based data against a set of constraints (so-called *shapes*). Importantly, SHACL allows to define *recursive shapes*, e.g., to state that children of persons must be persons. The recommendation left open the semantics of recursive shapes, but recently initial proposals to extend the official semantics to support recursion have emerged. These proposals are based on the principle of possibility (or, non-contradiction): a graph will be considered valid against a schema if one can *assign* shapes to nodes in a way that all the given shapes are satisfied. This semantics is *not constructive*, as it does not give guidelines how to obtain such an assignment, and it may lead to *unfounded assignments*, where the only reason to state that a node has a certain shape is because it serves to validate the graph.

In this paper we propose a stricter, more constructive SHACL semantics based on *stable models*, which are well-known in Answer Set Programming (ASP). Instead of just requiring non-contradictory assignments, we additionally require each shape assignment to be properly justified by the input constraints. We further exploit the connection to logic programming, and show that SHACL constraints can be naturally represented as logic programs, and that the validation problem for a graph and a SHACL schema can be represented as an ASP reasoning task. The proposed semantics is not only natural, but also enjoys computationally tractable validation in the presence of constraints with *stratified negation* (in contrast to the previous semantics). Finally, we also extend our semantics to *3-valued* stable models, which yields a more relaxed notion of validation that is tolerant to certain faults in the schema or data. By exploiting a connection between the existing 3-valued stable model semantics and the *well-founded* semantics for logic programs, we can use our translation into ASP to show a tractibility result, which we believe may lead to scalable, fault-tolerant validation services.

CCS CONCEPTS

• **Information systems → Semantic web description languages**; **Graph-based database models**.

KEYWORDS

SHACL, graph-structured data, answer set programming

ACM Reference Format:

Anonymous Author(s). 2020. Stable Model Semantics for Recursive SHACL. In *The Web Conference 2020*. ACM, New York, NY, USA, 11 pages. https://doi.org/TBA

The Web Conference 2020, April 20–24, 2020, Taipei Submission #1483

ACM ISBN TBA…$TBA
https://doi.org/TBA

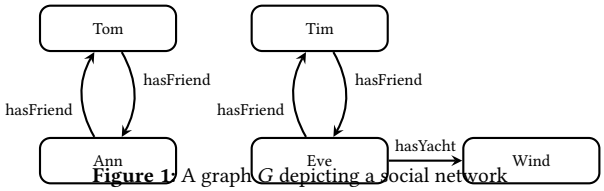

Figure 1: A graph G depicting a social network

1 INTRODUCTION

Constraints have traditionally been used to ensure quality of data in relational databases [3] and for semi-structured data [2]. Recently, constraints have also attracted considerable attention in the context of RDF data [8, 10, 24], and graph-structured data more generally [15, 16, 18, 20, 25]. Several systems already provide facilities for RDF validation (see, e.g., [14]), including commercial products.[1][2]

This created a need for standardizing a declarative language for RDF constraints, and for formal mechanisms to detect and describe violations of such constraints. SHACL, or Shapes Constraint Language[3], is one of the most important efforts in this direction, and has become a W3C recommendation in 2017. The specification provides a machine-readable language to specify constraints for data on the Web, and suggests algorithms to detect and describe constraint violations.

SHACL groups constraints in so-called "shapes" to be verified by certain nodes of the graph under validation, such that shapes may reference each other. As an example of how SHACL works, consider a social network such as that in Figure 1. Links are stored via a *hasFriend* relationship, and we depict other information about users in this network. We can now use SHACL to specify that all users in a network must be connected to at least one other user. We do it by constructing a shape *User* and specifying that all *User* nodes must satisfy the following constraint:

$$User \leftarrow (\exists hasFriend.\top)$$

This constraint is written in the abstract syntax proposed by Corman et al. [10], and specifies that each *User* must satisfy the query on the right hand side of the constraint, in this case, that there must be an edge of type *hasFriend*. In this paper we use this syntax for readability, but all these constraints can be transformed into SHACL constraints as defined in the official recommendation. To finish the SHACL schema, we must specify which of the nodes must be the *targets* of the shape *User*. In SHACL we can state particular nodes as targets, or one can state general constraints such as "all nodes of type 'person' must be *Users*". In this paper we also abstract the way in which targets are defined, assuming instead that one always has a set of nodes whose targets must be met, and which are expressed as atoms of the form $User(Ann)$, $User(Tim)$, etc.

[1] https://www.topquadrant.com/technology/shacl/
[2] https://www.stardog.com/docs/
[3] https://www.w3.org/TR/shacl/

An interesting aspect of SHACL is the usage of recursive constraints. For example, the constraint

$$Elite \leftarrow (\exists hasYacht.\top) \vee (\exists hasFriend.Elite)$$

defines another shape *Elite*, and specifies that a node complies with the shape *Elite* if it has a yacht, or befriends a node that complies with the shape *Elite*.

The semantics of recursive constraints is not straightforward, and in fact was left explicitly open in the SHACL specification. Let us come back to the graph in Figure 1, and suppose we want to validate this constraint over targets *Elite*(*Eve*), *Elite*(*Tim*) and *Elite*(*Ann*). *Eve* has a yacht, and therefore it is natural to expect that *Eve* can be assigned shape *Elite*, as this clearly satisfies the constraint. Further, with this information we can also safely expect *Tim* to be assigned shape *Elite*, as he is friend with *Eve*. But what about *Ann*?

Corman et al. [10] defined a semantics for SHACL based on the idea of *looking for a possible assignment of shapes to nodes*. According to their semantics, it is safe to say that the target *Elite*(*Ann*) is also satisfied, because there is an assignment of shapes to nodes in which all three targets are satisfied; we get that simply by assigning *Elite* to every node in the graph. We argue that this semantics is debatable: on what grounds can both *Tom* and *Ann* be assigned shape *Elite*? In contrast to *Eve* and *Tim*, where we base our assignment on the fact that *Eve* has a yacht, there is no justification for *Tom* and *Ann*.

We propose a stricter semantics: not only we need an assignment that satisfies all shape constraints, but we also require that such assignments must be well-founded, in the sense that there is a justification for every shape that is assigned to a node. To this end, we borrow the notion of a *stable model* from the *Answer Set Programming (ASP)* community, and define the stable model semantics for SHACL. We position all proposed semantics into a common framework, and show that stable model semantics fully agrees with the official recommendation when one restricts it to non-recursive constraints, and is a refinement of the semantics in [10].

Another benefit of our semantics is that the validation task can be encoded into an ASP program, that is data-independent, in the sense that the translation of the constraints does not need as input the graph being validated. Moreover, one can rely on off-the-shelf ASP solvers to perform validation. Indeed, our proof-of-concept implementation and evaluation show that for real RDF data this approach is feasible in practice.

We also define a notion of stratified SHACL schemas that generalizes the idea of ASP programs with stratified negation. For such schemas, our semantics has the advantage that the notion of validation is constructive: the validation problem for stratified schemas can be solved in polynomial time, and the results can be processed starting from the lower levels of the stratification and moving up. In contrast, the validation problem for the semantics in [10] remains NP-hard even for these stratified schemas.

Finally, we show that our framework can also be used to define *fault tolerant semantics*, where one may still validate some targets even if the schema presents consistency problems. Corman et al address this problem by using three-valued assignments where the value of some shapes in some nodes may be undetermined. We integrate this idea into our framework, and define a 3-valued stable model semantics that is again fault-tolerant but also imposes that shape assignments must be justified. Our framework gives us even more: we show that the complementary problem of checking whether the targets are satisfied in *all* 3-valued stable models can be solved in polynomial time as well.

An extended version of this paper containing additional details regarding proofs, the prototype implementation, and evaluation, are available in an anonymous repository[4].

2 GRAPH-STRUCTURED DATA AND SHACL CONSTRAINTS

In this section we define our basic notation of an RDF graph and the abstract syntax for SHACL. We abstract away from the concrete syntax of RDF, ignoring concrete domains (e.g., xsd:int) and term types (IRIs, literal, blank nodes), which are not relevant for the scope of this paper.

Definition 2.1 (Graph). Let $\mathbf{N}$ and $\mathbf{P}$ denote infinite, disjoint sets of *nodes* and *property names* respectively. A *graph* G is a finite set of atoms of the form $p(a, b)$, with $p \in \mathbf{P}$ and $a, b \in \mathbf{N}$. We use $V(G)$ to denote the set of nodes that appear in G.

For constraint expressions, we adopt the abstract syntax of SHACL "Core Constraint Components" (Section 4 of the SHACL specification [1]) proposed in [10].

Definition 2.2 (Shape expressions, constraints). Let $\mathbf{S}$ be an infinite set of *shape names*, disjoint from $\mathbf{N}$ and $\mathbf{P}$. A *path expression* is a regular expression E build from symbols in $\mathbf{P}$ using the usual operators $\cup$, $\cdot$ and $*$. A *shape expressions* ϕ is if the form:

$$\phi \ ::= \ \top \mid s \mid c \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid$$
$$\geq_n E.\phi \mid \ \leq_n E.\phi \mid \forall E.\phi \mid E = E'$$

where $s \in \mathbf{S}$, $c \in \mathbf{N}$, $n \in \mathbb{N}^+$ and E, E' are path expressions. For brevity, we sometimes write $\exists E.\phi$ instead of $\geq_1 E.\phi$.

A *shape constraint* is an expression of the form $s \leftarrow \phi$, with $s \in \mathbf{S}$ and ϕ a shape expression. If C is a set of constraints, we assume that for each shape name s that appears in C, there is exactly one ϕ such that $s \leftarrow \phi \in C$. We say that a shape name s *refers* to a shape name s' in C if $s \leftarrow \phi \in C$ and s' appears ϕ.

Because a shape expression ϕ may contain shape names, the evaluation of ϕ over a graph is defined w.r.t. to a so-called *shape assignment*, i.e. a labelling of the nodes of the graph with (sets of) shape names.

Definition 2.3 (Shape atom, assignment, decorated graph). A *shape atom* is an expression of the form $s(b)$, with $s \in \mathbf{S}$ and $b \in \mathbf{N}$. An *assignment* for a graph G is any set A of shape atoms s.t. $b \in V(G)$ for each $s(b) \in A$. If A is an assignment for G, then the set $I = G \cup A$ is called a *decorated graph*, and we let $V(I) = V(G)$.

For a decorated graph I, Table 1 defines the evaluation function $[\![\cdot]\!]^I$, which maps any path expression E to a binary relation $[\![E]\!]^I \subseteq V(I) \times V(I)$, and any shape expression ϕ to set of nodes $[\![\phi]\!]^I \subseteq V(I)$.

To complete the SHACL specification, we formalize the notions of *shape schema* and *target set*. We represent targets as shape atoms, where an atom of the form $s(b)$ represents the requirement for s to be assgined to (or, validated at) the node b of the input graph[5].

[4] https://bit.ly/33tG0Lk

[5] In practice, the SHACL specification allows to specify the targets of a shape via a monadic query, evaluated over the graph under validation.

$$[\![\top]\!]^I = V(I) \quad [\![c]\!]^I = \{c\} \text{ if } c \in V(I), \text{ and } \emptyset \text{ otherwise}$$
$$[\![p]\!]^I = \{(a,b) \mid p(a,b) \in I\} \quad [\![E^*]\!]^I = ([\![E]\!]^I)^*$$
$$[\![E \cup E']\!]^I = [\![E]\!]^I \cup [\![E']\!]^I \quad [\![E \cdot E']\!]^I = [\![E]\!]^I \circ [\![E']\!]^I$$
$$[\![s]\!]^I = \{b \mid s(b) \in I\} \quad [\![\neg\phi]\!]^I = V(I) \setminus [\![\phi]\!]^I$$
$$[\![\phi_1 \vee \phi_2]\!]^I = [\![\phi_1]\!]^I \cup [\![\phi_2]\!]^I \quad [\![\phi_1 \wedge \phi_2]\!]^I = [\![\phi_1]\!]^I \cap [\![\phi_2]\!]^I$$
$$[\![\forall E.\phi]\!]^I = \{a \mid \forall b : (a,b) \in [\![E]\!]^I \text{ implies } b \in [\![\phi]\!]^I\}$$
$$[\![\geq_n E.\phi]\!]^I = \{a \mid |\{(a,b) \in [\![E]\!]^I \text{ and } b \in [\![\phi]\!]^I\}| \geq n\}$$
$$[\![\leq_n E.\phi]\!]^I = \{a \mid |\{(a,b) \in [\![E]\!]^I \text{ and } b \in [\![\phi]\!]^I\}| \leq n\}$$
$$[\![E = E']\!]^I = \{a \mid \forall b : (a,b) \in [\![E]\!]^I \text{ iff } (a,b) \in [\![E']\!]^I\}$$

Table 1: Evaluation of shape expressions

Definition 2.4 (Shape schema). A *schema* is a pair $\langle C, T\rangle$, where C is set of shape constraints and T is a set of shape atoms.

In SHACL, the most important task is that of *validation*: given a graph G and a SHACL schema $\langle C, T\rangle$, is the graph valid for the schema? The specification states that G is valid when for each target $s(b)$ in T we have that b is a node of G, there is a constraint $s \leftarrow \phi$ in C and b conforms to ϕ. However, the SHACL specification leaves explicitly open the definition of "conforms" for recursive constraints. To address this case, [10] proposed to validate a graph if there exists an assignment A for G that complies with all constraints in C, and such that each target $s(b)$ is in A. We will use the term *supported-model semantics* to refer to this notion of validation.

Definition 2.5 (Models, supported models, validation). Given a set C of constraints, a decorated graph I is called a *model* of C if $[\![\phi]\!]^I \subseteq [\![s]\!]^I$ for all $s \leftarrow \phi \in C$. Moreover, I is a *supported model* of C if $[\![\phi]\!]^I = [\![s]\!]^I$ for all $s \leftarrow \phi \in C$.

A graph G is *valid* against a schema $\langle C, T\rangle$ under the *supported-model semantics*, if there exists an assignment A for G such that (i) $G \cup A$ is a supported model of C, and (ii) $T \subseteq A$.

3 STABLE MODEL SEMANTICS

In this section, we present a new semantics for SHACL constraints, which refines (by making it stronger) the semantics defined in [10]. The proposed semantics requires the validation of each target to have a well-founded justification. To see the importance of such justifications, recall the shape *Elite* and the constraint

$$Elite \leftarrow (\exists hasYacht.\top) \vee (\exists hasFriend.Elite)$$

defined in the Introduction, and the graph G in Figure 1. There are two ways of decorating this graph to obtain a supported model, and they correspond to the following assignments:

$$A_1 = \{Elite(Tim), Elite(Eve)\}$$
$$A_2 = \{Elite(Tom), Elite(Ann), Elite(Tim), Elite(Eve)\}$$

Thus, according to the supported model semantics, graph G would validate against the schema given by this single constraint and targets $\{Elite(Eve), Elite(Tim), Elite(Ann)\}$, because $G \cup A_2$ is a supported model that contains all of these facts.

We would like to define a semantics where the only models in which one may look for targets are those where all facts in assignments are justified by some constraint. Looking at the example above, it would be tempting to assume that all we need is to focus on minimal supported models. In fact, one can check that facts in models that are not minimal will never be justified by constraints. However, as the following example shows, this is not enough.

Example 3.1. Consider the following set C of shape constraints:

$$Elite \leftarrow (\exists hasYacht.\top) \vee (\exists hasFriend.Elite)$$
$$Moderate \leftarrow \neg Elite$$

Set C extends the constraints discussed in the introduction with an additional shape *Moderate*, and the constraint that all nodes that are not assigned *Elite* must be assigned shape *Moderate*. There are two supported models for C that extend G, they come from assignments

$$A'_1 = \{Moderate(Tom), Moderate(Ann), Elite(Tim), Elite(Eve)\}$$
$$A'_2 = \{Elite(Tom), Elite(Ann), Elite(Tim), Elite(Eve)\}$$

Again, using assignment A'_2 is debatable because the assignment of *Elite* to *Ann* or *Tom* is not justified. But the assignment A'_1 does satisfy all of our requirements. This time, however, both $G \cup A'_1$ and $G \cup A'_2$ are minimal supported models of C.

Our goal next is to define a semantics that prohibits unfounded justification cycles as observed in this example. To address this we draw inspiration from the stable model semantics for logic programs with negation. Intuitively, in a stable model of C every shape atom $s(b)$ must have a well-founded justification that involves an applicable constraint $s \leftarrow \phi \in C$. The shape expression ϕ at the node b might easily require the justification of further shape atoms. This process may lead to arbitrarily long justifications, but they are required to be well-founded, i.e. non-circular. Technically, this is achieved by means of a *level assignment*, which first assigns an integer (level) to every shape atom of a decorated graph, and which is then extended to assign levels to more complex shape expressions at different nodes of the graph. Intuitively, a stable model I of a constraint set C is then defined as a supported model of C, where additionally every shape atom $s(b)$ is justified by a constraint $s \leftarrow \phi \in C$ with the pair of ϕ and b assigned a strictly lower level than $s(b)$.

Let us make the above ideas more formal.

Definition 3.2. Let I be a decorated graph. A *level assignment* for I is a function level that maps tuples in $\{(\phi, a) \mid a \in [\![\phi]\!]^I\}$ to integers, and satisfies the following conditions:

(a) $\text{level}(\phi_1 \wedge \phi_2, a) = \max(\{\text{level}(\phi_1, a), \text{level}(\phi_2, a)\})$
(b) $\text{level}(\phi_1 \vee \phi_2, a) = \min(\{\text{level}(\phi_i, a) \mid i \in \{1,2\}, a \in [\![\phi_i]\!]^I\})$;
(c) $\text{level}(\geq_n E.\phi, a)$ is the smallest $k \geq 0$ for which there are n nodes $b_1, .., b_n$ such that $\text{level}(\phi, b_i) \leq k$, $(a, b_i) \in [\![E]\!]^I$ and $b_i \in [\![\phi]\!]^I$ for all $1 \leq i \leq n$;
(d) $\text{level}(\forall E.\phi, a) = \max(\{\text{level}(\phi, b) \mid (a,b) \in [\![E]\!]^I, b \in [\![\phi]\!]^I\})$.

We now define validation under stable model semantics, analogously to the supported model semantics of the previous section.

Definition 3.3 (Stable model semantics and validation). A decorated graph I is *stable model* of a set C of constraints, if

(i) I is a supported model of C, and
(ii) there exists a level assignment level such that: for all $s(a) \in I$, $\text{level}(\phi, a) < \text{level}(s, a)$, with $s \leftarrow \phi$ the constraint for s in C.

A graph G is valid against schema (C, T) under *stable model semantics*, if there exists an assignment A for G such that (i) $G \cup A$ is a stable model of C, and (ii) $T \subseteq A$.

We next illustrate how the proposed semantics handles the validation problem in Example 3.1.

Example 3.4. Take a level assignment including the following pieces, with $b \in \{Tom, Ann, Tim, Eve\}$:

$$\begin{aligned} \mathsf{level}(\top, b) &= 0 & \mathsf{level}(\exists hasYacht.\top, Eve) &= 0 \\ \mathsf{level}(Moderate, Tom) &= 0 & \mathsf{level}(\neg Elite, Tom) &= 0 \\ \mathsf{level}(Moderate, Ann) &= 0 & \mathsf{level}(\neg Elite, Ann) &= 0 \\ \mathsf{level}(Elite, Eve) &= 1 & \mathsf{level}(\exists hasFriend.Elite, Tim) &= 1 \\ \mathsf{level}(Elite, Tim) &= 2 \end{aligned}$$

Then for $G \cup A'_1$, this level assigment obeys condition (ii) in Definition 3.3, and thus $G \cup A'_1$ is a stable model of C. It is not difficult to see that $G \cup A'_2$ does not admit a level assignment, and thus is not a stable model of C.

4 TRANSLATION INTO LOGIC PROGRAMS

Our next goal is to show that SHACL constraints can be encoded as a logic program in such a way that the stable models of the input constraints are in one-to-one correspondence with the stable models of the resulting program, which opens the way to exploit existing logic programming engines to enable reasoning in SHACL. The translation is not only efficient (polynomial time), but also data-independent, which allows a single program resulting from input constraints to be used to validate multiple graphs.

4.1 Normal Form

Let us first define a normal form that will facilitate the presentation of the target translation.

Definition 4.1 (Normal form for SHACL). A SHACL constraint is in *normal form* if it has one of the following forms:

(NF1)	$s \leftarrow \top$	(NF2)	$s \leftarrow c$	(NF3)	$s \leftarrow s' \wedge s''$
(NF4)	$s \leftarrow s' \vee s''$	(NF5)	$s \leftarrow \neg s'$	(NF6)	$s \leftarrow E = E'$
(NF7)	$s \leftarrow \forall E.s'$	(NF8)	$s \leftarrow \geq_n E.s'$	(NF9)	$s \leftarrow \leq_n E.s'$

Here, $c \in \mathbf{N}$, and $s, s', s'' \in \mathbf{S}$.

The following enables us to focus on constraints in normal form.

Proposition 4.2. *A set C of constraints can be transformed in linear time into a set C' of constraints in normal form such that, for any graph G and target set T, G validates against (C, T) iff G validates against (C', T). This holds both for the supported-model semantics and the stable model semantics.*

Proof (Sketch). Assume a set C of constraints and a set T of targets. Take an arbitrary $s \leftarrow \phi \in C$. Suppose ϕ contains an occurrence of a complex subexpression ϕ', i.e., ϕ' occurs in ϕ, $\phi' \neq \phi$ and ϕ is not a shape name. Take a fresh shape name s^* that does not occur in C or T, replace the occurrence of ϕ' in ϕ by s^*, and add to C the constraint $s^* \leftarrow \phi'$, obtaining a constraint set C^*. It is not difficult to see that, if I is a stable model (resp., supported model) of C, then it can be extended to a stable model (resp., supported model) of C^* by adding shape atoms of the form $s^*(b)$. In particular, $I' = I \cup \{s^*(b) \mid b \in [\![\phi']\!]^I\}$ is a stable (resp., supported) model of C^*. On the other hand, if J is a stable (resp., supported) model of C^*, then $J \setminus \{s^*(b) \mid b \in V(J)\}$ is a stable (resp., supported) model of C. This trivially implies that C validates T iff C^* validates T.

By repeating the above rewriting at most linearly many times in the size of C, we can obtain a constraint set C' as described in the proposition statement. □

4.2 Answer Set Programming

We now recall the stable model semantics of logic programs [6].

We let $\mathsf{N}_{\mathsf{const}}$, $\mathsf{N}_{\mathsf{var}}$, and $\mathsf{N}_{\mathsf{pred}}$ be infinite, mutually disjoint sets of *constants* (like $a, b, c, \ldots$), *variables* (like $X, Y, Z, \ldots$), and *predicate symbols* (like $r, p, q, \ldots$), respectively. Elements of $\mathsf{N}_{\mathsf{const}} \cup \mathsf{N}_{\mathsf{var}}$ are also called *terms*. Each predicate symbol $p \in \mathsf{N}_{\mathsf{pred}}$ is associated to non-negative integer $ar(p)$, which is called the *arity* of p. A predicate symbol p is *n-ary* if $ar(p) = n$. If p is an n-ary predicate symbol, and $t_1, \ldots, t_n$ are terms, then the expression $p(t_1, \ldots, t_n)$ is called an *atom*. A rule r is an expression of the form

$$h \leftarrow b_1, \ldots, b_n, not\ b_{n+1}, \ldots, not\ b_m, \tag{1}$$

where $n, m \geq 0$, $h, b_1, \ldots, b_m$ are atoms. The atom h is called the *head* of r (denoted $head(r)$), the atoms $b_1, \ldots, b_n$ are called the *positive body atoms* of r, while $b_{n+1}, \ldots, b_m$ are the *negated body atoms* of r. We let $body^+(r) = \{b_1, \ldots, b_n\}$ and $body^-(r) = \{b_{n+1}, \ldots, b_m\}$. Each rule r as in (1) is required to be *safe*, i.e. every variable X that occurs in $h, b_{n+1}, \ldots b_m$ must also occur in some atom among $b_1, \ldots, b_n$. A program P is any finite set of rules. If Γ is a program or a rule with no occurrences of "*not*", then Γ is called *positive*. If Γ is a program, a rule or an atom with no occurrences of variables, then Γ is called *ground*. A ground rule of the form $h \leftarrow$, i.e. with no positive nor negated body atoms, is called a *fact*, and is often written simply as an atom h. A substitution σ is any partial function from $\mathsf{N}_{\mathsf{var}}$ to $\mathsf{N}_{\mathsf{const}}$. We denote by $\sigma(r)$ the rule obtained from a rule r by replacing every variable X for which $\sigma(X)$ is defined with $\sigma(X)$. We let $const(P)$ be the set of constants in P. For a program P, the *grounding* of P, denoted by $ground(P)$, is the set of ground rules r' such that $r' = \sigma(r)$ for some rule $r \in P$ and substitution σ s.t. $ran(\sigma) \subseteq const(P)$.

An *(Herbrand) interpretation* is any set of ground atoms. An interpretation I is a *model* of a ground positive rule $h \leftarrow b_1, \ldots, b_n$ if $\{b_1, \ldots, b_n\} \subseteq I$ implies $h \in I$. An interpretation I is a *model* of a ground positive program P if it is a model of every rule in P. As well known, any positive ground program P has a unique $\subseteq$-minimal model, denoted $LM(P)$ and called the *least model* of P. In particular, $LM(P)$ is a model of P, and there exists no $J \subsetneq I$ that is a model of P. The semantics of programs with negation is given using a program transformation due to Gelfond and Lifschitz [17]. Given an interpretation I and a program P, we define the *reduct* of P w.r.t. I as $P^I = \{head(r) \leftarrow body^+(r) \mid body^-(r) \cap I = \emptyset, r \in ground(P)\}$. An interpretation I is called a *stable model* (or *answer set*) of a program P if $I = LM(P^I)$.

Example 4.3. Let program P consist of the following rules

$$q(X) \leftarrow r(X), not\ p(X) \qquad w(X) \leftarrow q(X)$$
$$p(X) \leftarrow r(X), not\ q(X) \qquad r(a) \leftarrow$$

Observe that $I = \{r(a), p(a), w(a)\}$ is a model of P^I, but not the least model of P^I, since $\{r(a), p(a)\}$ is also a model of P^I. The only stable models of P are $\{r(a), p(a)\}$ and $\{r(a), q(a), w(a)\}$.

4.3 Encoding into ASP

In order to simplify the translation, but also for computational complexity reasons that will be explained later on, we modularize the encoding of SHACL constraints into ASP. In particular, we use a collection of "built-in" predicates to enrich an input graph. For a given graph G and a set of constraints C, these predicates allow us to flexibly navigate in G through the extension of each path expression E that appears in C. These predicates can be precomputed efficiently in advance (prior to resorting to an ASP reasoner), or one can use ASP rules (presented later) to express these predicates, obtaining a fully data-independent translation.

First, the unary **Dom** predicate allows us to access all nodes of an input graph G. Then, for a path expression E, we have five predicates that allow us to navigate the E-fillers of a given node in G. The binary predicate $\mathbf{Prop}_E$ stores all node pairs (a, b) such that there is a path from a to b in G that matches E. The unary predicate $\mathbf{Empty}_E$ captures nodes that do not have a filler for E, i.e., each node a for which there exists no node b such that $(a, b) \in \mathbf{Prop}_E$. The ternary predicate $\mathbf{Next}_E$ defines a successor relation over all E-fillers of a node a, namely $\mathbf{Next}_E(a, b, b')$ holds iff b, b' are E-fillers of a and b' is the successor of b. Finally, the binary predicates $\mathbf{First}_E$ and $\mathbf{Last}_E$ denote the first and the last E-fillers of a w.r.t. the successor relation induced by $\mathbf{Next}_E$. The formal definition of these predicates is given below.

Definition 4.4 (Built-in predicates). Assume a graph G and a set C of constraints. For a node a and a path expression E, let $range(E, a) = \{b \mid (a, b) \in [\![E]\!]^G\}$, let $\mathcal{E}_{E,a} = (b_1, \ldots, b_k)$ denote an arbitrary enumeration of $range(E, a)$, and let $\mathcal{F}_{E,a} = b_1$ and $\mathcal{L}_{E,a} = b_k$. Let G_C denote the smallest interpretation with $G \subseteq G_C$, and such that the following hold for each $a \in V(G)$ and path expression E in C:

(I) $\mathbf{Dom}(a) \in G_C$ for all $a \in V(G)$;
(II) $\mathbf{Prop}_E(a, b) \in G_C$ for all $b \in range(E, a)$;
(III) $\mathbf{Empty}_E(a) \in G_C$ if $range(E, a) = \emptyset$;
(IV) $\mathbf{Next}_E(a, b, b') \in G_C$ for all b, b' such that b' immiatly follows b in the enumeration $\mathcal{E}_{E,a}$, i.e. if $\mathcal{E}_{E,a} = (b_1, \ldots, b_k)$, then $b = i, b' = i + 1$ for some $1 \leq i < k$;
(V) $\mathbf{First}_E(a, b) \in G_C$ for $b = \mathcal{F}_{E,a}$;
(VI) $\mathbf{Last}_E(a, b) \in G_C$ for $b = \mathcal{L}_{E,a}$.

Definition 4.5 (Translation from SHACL to ASP). Let C be a set of constraints in normal form. We define the program P_C via the translation described in Table 2.

This translation is correct, in that it reduces validation against a SHACL schema (under stable model semantics) to checking whether some atoms belong to some stable models of an ASP program.

Constraint α	**ASP rules**
$s \leftarrow \top$	$s(X) \leftarrow \mathbf{Dom}(X)$
$s \leftarrow c$	$s(c) \leftarrow \mathbf{Dom}(c)$
$s \leftarrow s'$	$s(X) \leftarrow s'(X)$
$s_3 \leftarrow s_1 \wedge s_2$	$s_3(X) \leftarrow s_1(X), s_2(X)$
$s_3 \leftarrow s_1 \vee s_2$	$s_3(X) \leftarrow s_1(X) \qquad s_3(X) \leftarrow s_2(X)$
$s \leftarrow \neg s'$	$s(X) \leftarrow \mathbf{Dom}(X), not\ s'(X)$
$s \leftarrow E = E'$	$s^\alpha(X) \leftarrow R_E(X, Y), not\ R_{E'}(X, Y)$ $s^\beta(X) \leftarrow R_{E'}(X, Y), not\ R_E(X, Y)$ $s(X) \leftarrow \mathbf{Dom}(X), not\ s^\alpha(X), not\ s^\beta(X)$
$s \leftarrow \forall E.s'$	$s(X) \leftarrow \mathbf{Empty}_E(X)$ $u^\alpha(X, Y) \leftarrow \mathbf{First}_E(X, Y), s'(Y)$ $u^\alpha(X, Z) \leftarrow \mathbf{Next}_E(X, Y, Z), u^\alpha(X, Y), s'(Z)$ $s(X) \leftarrow \mathbf{Last}_E(X, Y), u^\alpha(X, Y)$
$s \leftarrow \geq_n E.s'$	$s(X) \leftarrow AtLeast^n_{E,s'}(X, Y)$
$s \leftarrow \leq_n E.s'$	$s(X) \leftarrow not\ AtLeast^{n+1}_{E,s'}(X, Y)$
The following $n + 1$ rules are added for each integer n, path expression E, and a shape name s' that appear in a constraint of the form $s \leftarrow \geq_n E.s'$	
	$AtLeast^0_{E,s'}(X, X) \leftarrow \mathbf{Dom}(X)$ $AtLeast^1_{E,s'}(X, Y) \leftarrow \mathbf{First}_E(X, Y), s'(Y)$ $AtLeast^i_{E,s'}(X, Z) \leftarrow \mathbf{Next}_E(X, Y, Z), AtLeast^i_{E,s'}(X, Y)$ $AtLeast^{i+1}_{E,s'}(X, Z) \leftarrow \mathbf{Next}_E(X, Y, Z), AtLeast^i_{E,s'}(X, Y), s'(Z)$

Table 2: Translation from SHACL to ASP rules

Proposition 4.6. *Let G be a graph and (C, T) a schape schema. Then G is valid against (C, T) under stable model semantics iff $P_C \cup G_C$ has a stable model I such that $T \subseteq I$.*

Proof (Sketch). We first observe that a graph is valid against a non-normalized schema iff it is valid against its normalized version. So we focus w.l.o.g. on the case where where C is normalized. ($\Rightarrow$). If G is valid against (C, T), then there is a stable model $I = G \cup A$ of (G, C) s.t. $T \subseteq I$. We let I^+ denote $G_C \cup A$. Then $T \subseteq I^+$. Now let R denote the reduct of $P_C \cup G_C$ wrt I^+. We show that I^+ is a stable model of $P_C \cup G_C$, i.e. I^+ and the (unique) smallest model $LM(R)$ of R coincide. First, from the translation provided in Table 2, it can be easily seen that I^+ is a model of $P_C \cup G_C$, and therefore also a model of R, so that $LM(R) \subseteq I^+$. Then to show that $I^+ \subseteq LM(R)$, we use an alternative definition of $LM(R)$, as the smallest set B that verifies (i) $G_C \subseteq B$, and (ii) if $body^+(r) \subseteq B$ for some rule $r \in R$ with $head(r) = s(a)$, then $s(a) \in B$. Now since I is a stable model of (G, C), there must be a function level_I that satisfies the requirements of Definition 3.2. We consider all integers k s.t. $\text{level}(s, a) = k$ for some $s(a) \in A$, and we order them as $k_1, .., k_n$, so that $k_i < k_{i+1}$.

Then by induction on $i \in \{1..n\}$, it can be shown that $s(a) \in B$ must hold for each $s(a)$ such that $\text{level}(s,a) = k_i$, based on Definition 3.2, and the fact that $\text{level}(s,a) > \text{level}(\phi,a)$ must hold if $s \leftarrow \phi$ is the constraint for s in C.

($\Leftarrow$). Let I be a stable model of $P_C \cup G_C$ such that $T \subseteq I$. We let I^- denote $G \cup A$. Then $T \subseteq I^-$. We show that I^- is a stable model of C. First, from the translation provided in Table 2, it can be easily seen that I^- is a model of C. Then we define a function level_{I^-} that satisfies the requirements of Definition 3.2, as follows. Let R denote the reduct of $P_C \cup G_C$ w.r.t. I. Then let $A_0, .., A_n$ be the partition of A defined by (i) $A_0 = \emptyset$, (ii) $A_i = \{s(a) \mid r \in R, s(a) = \mathit{head}(r), \mathit{body}^+(r) \subseteq G_C \cup A_0 \cup .. \cup A_{i-1}\}$. And for any $a \in V(G)$ and path expression E, let $\mathit{qRange}(E,a,s) = \{b \in \mathit{range}(E,a) \mid s(b) \in A\}$. To define level_{I^-}, we first set $\text{level}(s,a) = i$ if $s(a) \in A_i$ for some $i \in \{1..n\}$. Then we extend level_{I^-} to (some) other formulas, for each $a \in V(G)$, as follows: (i) $\text{level}(\top,a) = 0$ (ii) $\text{level}(\neg s,a) = 0$ if $s(a) \notin A$ (iii) $\text{level}(c,a) = 0$ if $c = a$ (iv) $\text{level}(s_1 \wedge s_2,a) = \max(\{\text{level}(s_1,a), \text{level}(s_2,a)\})$ if $s_1(a), s_2(a) \in A$ (v) $\text{level}(s_1 \vee s_2,a) = \min(\{\text{level}(s_1,a), \text{level}(s_2,a)\})$ if $s_1(a) \in A$ or $s_2(a) \in A$ (vi) $\text{level}(\forall E.s,a) = \max(\{\text{level}(s,b) \mid b \in \mathit{range}(E,a)\})$ if $\mathit{qRange}(E,a,s) = \mathit{range}(E,a)$ (vii) if $|\mathit{qRange}(E,a,s)| \geq n$, then $\text{level}(\geq_n E.s,a)$ is the smallest $k \geq 0$ for which there are $a_1, .., a_n \in \mathit{qRange}(E,a,)$ s.t. $\text{level}(s,a_i) \leq k$.

Finally, if ϕ is a shape expression, we let $\text{cl}(\phi)$ designate the set of formulas that can be built (recursively) by replacing some shape names in ϕ by their respective definitions in C. And if $\text{level}(\phi,a)$ is defined, for each $\phi' \in \text{cl}(\phi)$, we set $\text{level}(\phi',a) = \text{level}(\phi,a)$. □

4.4 Stratified Programs and Complexity

It is known that basic reasoning in ASP is not tractable in terms of computational complexity. E.g., the task of checking the existence of a stable model for a ground program P is an NP-complete problem (see, e.g., the survey [11]). Intuitively, intractability may arise when a program describes recursion that involves negation. This may cause a program to have multiple stable models, or no stable model at all. Prohibiting such recursion leads to the so-called *stratified programs* (or *Datalog with stratified negation*). These always have a unique stable model, which can be computed efficiently. Let us define this fragment for ASP first, we will later use it to identify a computationally well-behaved fragment of SHACL constraints.

Definition 4.7 (Stratified program). A program P is called *stratified* if it can be partitioned into programs $P_0, \ldots, P_k$ such that the following holds for every $0 \leq i \leq k$:

(a) if $i < k$ and $p(\vec{t}) \in \mathit{body}^+(r)$ for some rule $r \in P_i$, then p does not occur in the head of any rule in $P_{i+1} \cup \ldots \cup P_k$;
(b) if $p(\vec{t}) \in \mathit{body}^-(r)$ for some some rule $r \in P_i$, then p does not occur in the head of any rule in $P_i \cup \ldots \cup P_k$.

The sequence $P_0, \ldots, P_k$ as above is called a *stratification* of P.

Assume a stratification $P_0, \ldots, P_k$ of a program P, and consider a program P_i with $0 \leq i \leq k$. Intuitively, condition (a) requires that the extensions of body atoms can only be populated by the rules at the lower or current "stratum", i.e., by the rules in $P_0 \cup \ldots \cup P_i$. Condition (b) makes this requirement stronger: the extensions of negated atoms must be fully defined by the rules at the strictly lower "stratum". Together (a) and (b) ensure that a stable model of P can be gradually built in polynomial time in $k+1$ steps by computing the least models of $k+1$ positive programs derived from $P_0, \ldots, P_k$. In particular, let $I_1, \ldots, I_k$ be interpretations defined as follows: (i) $I_0 = LM(P_0^{\emptyset})$, (ii) $I_n = I_{n-1} \cup LM(P_n^{I_{n-1}})$, for $0 < n \leq k$. Then I_K is known to be the unique stable model of P.

We now define a notion of stratification for constraints, which closely resembles the one above.

Definition 4.8 (Stratified shape constraints). A shape name s is said to occur *negatively* in a shape expression ϕ if s occurs in ϕ in a subexpression of the form $\neg\phi'$ or $\leq_n E.\phi'$. We say that a shape name s is *defined* in a set C of constraints if $s \leftarrow \phi \in C$ for some shape expression ϕ.

A set C of constraints is called *stratified* if it can be partitioned into sets $C_0, \ldots, C_k$ such that the following hold for $0 \leq i \leq k$:

(a) if $i < k$ and s' occurs in ϕ for some $s \leftarrow \phi \in C_i$, then s' is not defined in $C_{i+1} \cup \ldots \cup C_k$;
(b) if s' occurs negatively in ϕ for some $s \leftarrow \phi \in C_i$, then s' is not defined in $C_i \cup \ldots \cup C_k$.

A set of constraints is *stratified* if it admits a stratification. By extension, we say that a schema (C,T) is stratified if C is stratified.

The set of constraints in Example 3.1 are for instance stratified. Let

$$C_1 = \{\mathit{Elite} \leftarrow (\exists \mathit{hasYacht}.\top) \vee (\exists \mathit{hasFriend}.\mathit{Elite})\}, \text{ and}$$
$$C_2 = \{\mathit{Moderate} \leftarrow \neg \mathit{Elite}\}.$$

The conditions above are satisfied, since *Elite* appears negatively in C_2 but is defined only in C_1.

We can now make an interesting connection between stratified ASP programs and stratified constraints. First we observe that constraint normalization preserves stratification. Then it can be easily seen that the translation of a normalized set of constraints into an ASP program provided in Section 4.3 also preserves this property.

Lemma 4.9. *If C is a stratified set of constraints in normal form, then P_C is a stratified program.*

Proof (Sketch). Assume an arbitrary stratified set C of constraints in normal form, and consider P_C Let $C_0, \ldots, C_k$ be a stratification of C. It is not difficult to check that $P_{C_0}, \ldots, P_{C_k}$ is a stratification of P_C. □

Finally, it is also clear that the translation to P_C and G_C^+ is polynomial in both the size of C and the graph G. Together with Proposition 4.6, this leads to one of the main results of this paper: validation of stratified constraints under stable model semantics is tractable:

Proposition 4.10. *Deciding if a graph G is valid against a stratified schema (C,T) under stable model semantics is PTIME-complete.*

Proof (Sketch). The upper bound follows from the ASP translation and the complexity of reasoning with stratified programs [11].

For the lower bound, we reduce from the problem of entailment in positive ground programs. Fix an arbitrary ground atom g. It is a PTIME-complete problem to decide, given a ground positive program P, whether $g \in LM(P)$ holds. Assume a ground positive program P. W.l.o.g., we can assume that g appears in P. From P we construct a graph G_P as follows. The set $V(G_P)$ is the set of nodes c_γ, where γ is a ground atom or a ground rule that appears P.

We use two property names p_{rule}, p_{atom}. For every rule γ in P with the form $h \leftarrow b_1, \ldots, b_n$, we add $p_{rule}(c_{b_1}, c_\gamma), \ldots, p_{rule}(c_{b_n}, c_\gamma)$ to G_P. For every atom h in P and every rule γ in P with $head(\gamma) = h$, we add $p_{atom}(c_\gamma, c_h)$ to G_P. Take a set of constraints $C_P = \{s \leftarrow \exists p_{atom}.s;\ s \leftarrow \forall p_{rule}.s\}$, and let $T_P = \{s(c_g)\}$. It is not difficult to see that $g \in LM(P)$ iff G_P validates against (C_P, T_P). □

This strongly contrasts with validation w.r.t. supported models, which was shown to be NP-hard in the size of the graph for stratified constraints [10] . We also observe that the definition of stratified constraints adopted in this article is less restrictive than the one used in Corman et al. [10]. Indeed we used $\forall$ and $\vee$ as native operators of the shape expression language, whereas Corman et al. only argue that they can be expressed by means of $\neg$ (combined with $\wedge$ and $\geq_n$). So our notion of stratification identifies a larger fragment of SHACL as computationally well-behaved (when evaluated under stable model semantics).

We now justify the restriction put above on the usage of the $\leq_n$ operator, when defining stratification for a set of constraints (Definition 4.8). Indeed, without such restriction, validation under stable model semantics becomes intractable, even if the constraints are negation-free.

Proposition 4.11. *Deciding if a graph G is valid against a schema (C, T) under stable model semantics is NP-hard if C contains expressions of the form $\leq_n E.\phi$, and none of the form $\neg\phi$.*

Proof (Sketch). We provide a polynomial time reduction from 3SAT. Assume a 3CNF formula $\varphi = (L_1^1 \vee L_2^1 \vee L_3^1) \wedge \ldots \wedge (L_1^k \vee L_2^k \vee L_3^k)$. We build a graph G and schema (C, T) such that φ is satisfiable iff G is valid against (C, T). For every propositional variable v in φ, we use two shape names $v^{\mathrm{t}}, v^{\mathrm{f}}$, and we let $sh(v) = v^{\mathrm{t}}$ and $sh(\neg v) = v^{\mathrm{f}}$. We further use the shape names sat and $cl_1, \ldots, cl_k$. We set $G = \{r(a, a)\}$ and $T = \{sat(a)\}$. The set of constraints C is built as follows. First, for each variable v in φ, C contains the constraints $v^{\mathrm{t}} \leftarrow \leq_0 r.v^{\mathrm{f}}$ and $v^{\mathrm{f}} \leftarrow \leq_0 r.v^{\mathrm{t}}$. Intuitively, they implement a guess of a truth assignment for the variables in φ. To check that the assignment causes φ to evaluate to "true", we add to C the constraints $sat \leftarrow cl_1 \wedge \cdots \wedge cl_k$, and $cl^i \leftarrow sh(L_1^i)$ for all $1 \leq i \leq k$, and all $j \in \{1, 2, 3\}$. □

We conclude by observing that validation under the stable model semantics is not harder in the worst case than validation under the supported model semantics, for arbitrary (i.e. not necessarily stratified) constraints. This follows immediately from the fact that the translation is polynomial, Proposition 4.6 and the known upper bounds for reasoning in ASP under stable model semantics [11].

Proposition 4.12. *Deciding whether a graph G is valid against a schema (C, T) under the stable model semantics is NP-complete.*

The NP lower bound above follows immediately because the setting subsumes prositional ASP programs, for which deciding the existence of a stable model is NP-hard [11].

4.5 Expressing Built-in Predicates using Rules

We now show that the built-in predicates mentioned in Section 4.3 can also be easily encoded as ASP rules. This offers the flexibility to delegate the evaluation of these predicates to an ASP solver, in addition to constraint validation *per se*. The encoding is provided in Table 3, where the points (I-VI) are in correspondence with the items in Definition 4.4. The rules in (⋆) are standard rules that help in defining successor functions, as required in point (IV). Given a set C of constraints, the table describes a program B_C such that, given any graph G, any stable model I of $B_C \cup G$ corresponds to a graph G_C as described in Definition 4.4.

(I)	$\mathbf{Dom}(X) \leftarrow p(X, Y)$
	$\mathbf{Dom}(Y) \leftarrow p(X, Y)$
(II)	$\mathbf{Prop}_p(X, Y) \leftarrow p(X, Y)$
	$\mathbf{Prop}_{E \cdot E'}(X, Y) \leftarrow \mathbf{Prop}_E(X, Y), \mathbf{Prop}_{E'}(Y, Z)$
	$\mathbf{Prop}_{E \cup E'}(X, Y) \leftarrow \mathbf{Prop}_E(X, Y)$
	$\mathbf{Prop}_{E \cup E'}(X, Y) \leftarrow \mathbf{Prop}_{E'}(X, Y)$
	$\mathbf{Prop}_{E^*}(X, X) \leftarrow \mathbf{Dom}(X)$
	$\mathbf{Prop}_{E^*}(X, Z) \leftarrow \mathbf{Prop}_{E^*}(X, Y), \mathbf{Prop}_E(Y, Z)$
(III)	$\mathbf{NonEmpty}_E(X) \leftarrow \mathbf{Prop}_E(X, Y)$
	$\mathbf{Empty}_E(X) \leftarrow \mathbf{Dom}(X), \mathit{not}\ \mathbf{NonEmpty}_E(X)$
(⋆)	$\mathbf{LEQ}_E(X, Y, Y) \leftarrow \mathbf{Prop}_E(X, Y)$
	$\mathbf{LEQ}_E(X, Y, Y') \leftarrow \mathbf{Prop}_E(X, Y), \mathbf{Prop}_E(X, Y'), \mathit{not}\ \mathbf{LEQ}_E(X, Y', Y)$
	$\mathbf{LEQ}_E(X, Y, Z) \leftarrow \mathbf{LEQ}_E(X, Y, W), \mathbf{LEQ}_E(X, W, Z)$
	$\mathbf{EQ}(X, X) \leftarrow \mathbf{Dom}(X)$
	$\mathbf{Distant}_E(X, Y, Z) \leftarrow \mathbf{LEQ}_E(X, Y, W), \mathbf{LEQ}_E(X, W, Z), \mathit{not}\ \mathbf{EQ}(Y, W), \mathit{not}\ \mathbf{EQ}(Z, W)$
(IV)	$\mathbf{Next}_E(X, Y, Y') \leftarrow \mathbf{LEQ}_E(X, Y, Y'), \mathit{not}\ \mathbf{EQ}(Y, Y'), \mathit{not}\ \mathbf{Distant}_E(X, Y, Y')$
	$\mathbf{NonFirst}_E(X, Y') \leftarrow \mathbf{Next}_E(X, Y, Y')$
	$\mathbf{NonLast}_E(X, Y) \leftarrow \mathbf{Next}_E(X, Y, Y')$
(V)	$\mathbf{First}_E(X, Y) \leftarrow \mathbf{Prop}_E(X, Y), \mathit{not}\ \mathbf{NonFirst}_E(X, Y)$
(VI)	$\mathbf{Last}_E(X, Y) \leftarrow \mathbf{Prop}_E(X, Y), \mathit{not}\ \mathbf{NonLast}_E(X, Y)$

Table 3: Encoding of the built-in predicates using rules. For an input set C of constraints, p ranges over all property names that appear in C, and the definition of predicates with a path expression in the subscript is limited to expressions that appear in C.

4.6 Encoding of Supported Models

We now show that the translation from Table 2 can also be extended to capture supported model semantics. We note that such an encoding cannot preserve stratification in general (unless P ⊆ NP), because validation under supported models is NP-complete.

Consider a graph G and a set of constraints C in normal form. For each shape s defined in C, we introduce a predicate s_{neg} that validates where s does not (and conversely). To this end, we introduce two rules:

$$s(X) \leftarrow \neg s_{neg}(X), \mathbf{Dom}(X). \quad s_{neg}(X) \leftarrow \neg s(X), \mathbf{Dom}(X). \quad (2)$$

For each shape s defined in C, we also introduce a predicate s^* and two rules to discard all guessed non-necessary assignments (direction $[\![\phi]\!]^I \supseteq [\![s]\!]^I$ of Def. 2.5):

$$s^*(X) \leftarrow \mathrm{def}(s). \quad \bot \leftarrow s(X), \neg s^*(X). \tag{3}$$

Here, $\mathrm{def}(s)$ is the encoding of ϕ given by Table 2 for $s \leftarrow \phi \in C$. We denote above rules as the program P_C^{2v}.

Proposition 4.13. *Let G be a graph and (C, T) a shape schema. Then the following are equivalent:*

- *G is valid against (C, T) under supported-model semantics.*
- *$G_C \cup P_C \cup P_C^{2v}$ has a stable model I such that $T \subseteq I$.*

5 3-VALUED ASSIGNMENTS

Corman et al. also argue that the supported-model semantics we recalled in Section 2 may be too restrictive in the presence of non-stratified schemas [10].

Example 5.1. As an illustration, consider the following example, which violates the stratification conditions in Definition 4.8:

$$C = \{s_1 \leftarrow \exists r_1.\top;\ s_2 \leftarrow \exists r_2.\neg s_2\}$$
$$T = \{s_1(a)\}$$
$$G = \{r_1(a,b), r_2(b,b)\}$$

In this example, the only target is $s_1(a)$, and the constraint for s_1 only requires that a node must have an r_1-successor. Then since a has an r_1-successor (namely b) in the graph G under validation, one would expect the target to be valid. But then notice that the constraint for s_2 can never be satisfied for any node, no matter if we assign shape name s_2 to that node or not. This means that there are no supported models for the schema (C, T), and therefore G does not validate it under supported or stable model semantics.

To address this issue, Corman et al. propose using *partial* assignments, in which some assignments from shapes to nodes are left undetermined, effectively moving from a *2-valued* to a *3-valued* setting. We now recall this semantics, using the notation of Section 2.

Definition 5.2 (shape literal, literal assignment). A *shape literal* is an expression of the form $s(a)$ or $0.5s(a)$, where $s(a)$ is a shape atom. A set B of shape literals is *consistent* if there is no shape atom $s(a)$ such that $s(a) \in B$ and $0.5s(a) \in B$. A *literal assignment* for a graph G is any consistent set of shape literals $A \subseteq \{s(a), 0.5s(a) \mid a \in V(G)\}$. If A is a literal assignment for G, then the set $I = G \cup A$ is called a *literal-decorated graph*.

Intuitively, a shape atom $0.5s(a)$ states that the truth value of shape name s at node a is undetermined. We now define a function $[\phi]^{a,I}$ which assigns to every complex shape expression ϕ its truth value at node a in I. The inductive definition of $[\phi]^{a,I}$ is an immediate extension Kleene's 3-valued logic, and is given in Table 4. Note that in case I is only 2-valued (i.e., if it does not contain atoms of the form $0.5s(a)$), then $[\cdot]^{a,I}$ and $[\![\cdot]\!]^I$ collapse, i.e. we have $[\phi]^{a,I} \in \{0, 1\}$ and $[\![\phi]\!]^I = \{a \in V(G) \mid [\phi]^{a,I} = 1\}$.

Validation via 3-valued assignments introduced in [10] can be formulated analogously to validation via 2-valued assignments in Section 2: first define 3-valued supported models, and then consider a graph valid iff there exists a 3-valued supported model of the graph and constraints that contains all targets.

$$\begin{aligned}
[\top]^{a,I} &= 1 \\
[c]^{a,I} &= \begin{cases} 1 \text{ if } a = c, \\ 0 \text{ otherwise} \end{cases} \\
[s]^{a,I} &= \begin{cases} 1 \text{ if } s(a) \in I \\ 0.5 \text{ if } 0.5s(a) \in I \\ 0 \text{ otherwise} \end{cases} \\
[\neg\phi]^{a,I} &= 1 - [\phi]^{a,I} \\
[\phi_1 \wedge \phi_2]^{a,I} &= \min(\{[\phi_1]^{a,I}, [\phi_2]^{a,I}\}) \\
[\phi_1 \vee \phi_2]^{a,I} &= \max(\{[\phi_1]^{a,I}, [\phi_2]^{a,I}\}) \\
[E_1 = E_2]^{a,I} &= \begin{cases} 1 \text{ if } \{b \mid (a,b) \in [\![E_1]\!]^I\} = \{b' \mid (a,b') \in [\![E_2]\!]^I\} \\ 0 \text{ otherwise} \end{cases} \\
[\geq_n E.\phi]^{a,I} &= \begin{cases} 1 \text{ if } |\{b \mid (a,b) \in [\![E]\!]^I \text{ and } [\phi]^{b,I} = 1\}| \geq n \\ 0 \text{ if } |\{b \mid (a,b) \in [\![E]\!]^I\}| - \\ \qquad |\{b \mid (a,b) \in [\![E]\!]^I \text{ and } [\phi]^{b,I} = 0\}| < n \\ 0.5 \text{ otherwise} \end{cases} \\
[\forall E.\phi]^{a,I} &= 1 - [\geq_1 E.\neg\phi]^{a,I}
\end{aligned}$$

Table 4: 3-valued evaluation of expression ϕ at node a in interpretation I

Definition 5.3 (3-valued supported models, validation). A literal-decorated graph $I = G \cup A$ is a *3-valued supported model* of a set C of constraints if for each $s \leftarrow \phi \in C$ and for each node $a \in V(G)$ (i) A contains $s(a)$ iff $[\phi]^{a,I} = 1$, and (ii) A contains $0.5s(a)$ iff $[\phi]^{a,I} = 0.5$.

A graph G is *valid* against a schema (C, T) under *3-valued supported-model semantics* if there exists a literal assignment A such that (i) $G \cup A$ is a 3-valued supported model of C, and (ii) $T \subseteq A$.

Example 5.4 (Example 5.1 continued). The graph G has a 3-valued supported model of C, given by the assignment $A = \{s_1(a), 0.5s_2(b)\}$. The assignment A contains all shape atoms in T, hence G is valid against (C, T) under 3-valued supported-model semantics.

5.1 3-valued Stable Models

Even if 3-valued supported models give us flexibility to leave assignments undetermined and focus only on what is needed to verify the targets, this semantics suffers from the same problems we discussed for the 2-valued supported-model semantics in Section 3. Thus, it would be desirable to define an analog to 2-valued stable model semantics that incorporates the idea of 3-valued assignments. As it turns out, our definition of assignments lets us extend our level-based definition to the 3-valued case: all we need now is that undetermined (0.5) literals must also be justified.

Definition 5.5. Let I be a literal-decorated graph. A *level assignment* for I is a function level that maps tuples in $\{(\phi, a) \mid a \in [\phi]^{a,I} > 0\}$ to integers, and satisfies the following conditions:

a) $\mathrm{level}(\phi_1 \wedge \phi_2, a) = \max(\{\mathrm{level}(\phi_1, a), \mathrm{level}(\phi_2, a)\})$
b) $\mathrm{level}(\phi_1 \vee \phi_2, a) = \min(\{\mathrm{level}(\phi_i, a) \mid i \in \{1, 2\}, [\phi_i]^{a,I} = t\})$, where $t = [\phi_1 \vee \phi_2]^{a,I}$
c) $\mathrm{level}(\geq_n E.\phi, a)$ is the smallest $k \geq 0$ for which there exist n nodes $c_1, \ldots, c_n$ such that
   (a) $[\phi]^{c_1,I} \geq [\geq_n E.\phi]^{a,I}, \ldots, [\phi]^{c_n,I} \geq [\geq_n E.\phi]^{a,I}$, and
   (b) $(a, c_i) \in [\![E]\!]^I$ and $\mathrm{level}(\phi, c_i) \leq k$ for all $1 \leq i \leq n$;
d) $\mathrm{level}(\forall E.\phi, a) = \max(\{\mathrm{level}(\phi, b) \mid (a,b) \in [\![E]\!]^I, b \in [\![\phi]\!]^I\})$.

We can now define 3-valued stable models and the corresponding notion of validation:

Definition 5.6 (3-valued stable models). A literal-decorated graph I is a *3-valued stable model* of a set C of constraints if

(i) I is a 3-valued supported model of C, and
(ii) there exists a level assignment level such that: for all $s(a) \in I$, $\mathsf{level}(\phi, a) < \mathsf{level}(s, a)$ with $s \leftarrow \phi$ the constraint for s in C.

Definition 5.7 (Validation in 3-valued stable models). A graph G is valid against a schema (C, T) under *3-valued stable model semantics* if there exists a literal assignment A for G such that (i) $G \cup A$ is a 3-valued stable model of C, and (ii) $T \subseteq A$.

Example 5.8 (Example 5.1 continued). Consider again the graph G and schema (C, T) from Example 5.1, and the assignment $A = \{s_1(a), 0.5s_2(b)\}$. To show that $I = G \cup A$ is a stable model of C, we use a level assignment level containing the following parts:

$$\mathsf{level}(\top, a) = 0 \qquad \mathsf{level}(\top, b) = 0$$
$$\mathsf{level}(\geq_1 r_1.\top, a) = 0 \qquad \mathsf{level}(s_1, a) = 1$$
$$\mathsf{level}(\neg s_2, b) = 0 \qquad \mathsf{level}(\geq_1 r_2.\neg s_2, a) = 0$$
$$\mathsf{level}(s_2, b) = 1$$

The notion of 3-valued stable models has also been studied for ASP programs. We use the notion from [22] (we note that the use of 3 values to relax the semantics of a logic program in order to draw reasonable conclusions from *incoherent* programs has a long tradition in the logic programming community; see, e.g., [4, 13, 19, 23]). With this notion, we can provide a proof of concept for our semantics: $G \cup A$ is a 3-valued stable model of a normalized set C of constraints iff $G_C \cup A$ is a 3-valued stable model of the ASP encoding P_C of C. So validation under 3-valued stable model semantics can be characterized in terms of 3-valued stable models for ASP programs:

Proposition 5.9. *A graph G is valid against a shape schema (C, T) under 3-valued stable model semantics iff $P_C \cup G_C$ has a 3-valued stable model I such that $T \subseteq I$.*

Proof (Sketch). The argument is almost identical to the one for Proposition 4.6: instead of the standard 2-valued semantics of ASP and the 2-valued semantics of SHACL used in Proposition 4.6, we use the standard 3-valued semantics of ASP [22] and the 3-valued stable model semantics defined above. □

5.2 Cautious Validation with 3-valued Stable Models

In their efforts to lift the SHACL semantics to recursive constraints, Corman et. al only focused on what can be seen as *brave* validation: all we need is one supported model that contains all targets. However, it is also natural to study *cautious* validation, which corresponds to validating those targets that are valid in all supported models. In logical terms, this corresponds to entailment. Of course, cautious validation can also be defined for 2-valued stable model semantics, or 3-valued stable model semantics.

A key advantage of the 3-valued stable model semantics defined above is that it enjoys tractable algorithms for cautious validation. This can be seen by exploiting the link between 3-valued stable models of logic programs and the *well-founded semantics* of logic programs (see, e.g., [7, 26] for more details on the well-founded semantics, and [22] for the connection to the 3-valued stable model semantics of logic programs). Let us begin with the formal definitions of cautious validation.

Definition 5.10. A graph G is valid against a schema (C, T) under *cautious 3-valued stable model semantics* if $T \subseteq A$ for each literal assignment A for G such that $G \cup A$ is a 3-valued stable model of C.

Example 5.11. Consider a set C with the following constraints:

$$\mathit{CurrentStudent} \leftarrow \neg \mathit{PastStudent} \vee \exists \mathit{hasStudID}.\top$$
$$\mathit{PastStudent} \leftarrow \neg \mathit{CurrentStudent} \vee \exists \mathit{hasUnenrollDate}.\top$$

Intuitively, a person should be considered as currently a student in case s/he has a student ID, or one cannot infer that s/he is a former student. The second constraint tells us that a person should be considered a former student if s/he has unenrolled, or it is not possible to validate that the person is currently a student. Consider the graph G with the following facts:

$$\mathit{hasStudID}(\mathit{Ann}, \mathit{123}),\ \mathit{hasUnenrollDate}(\mathit{Bob}, \mathit{1.1.19}),\ \mathit{Person}(\mathit{Eve}).$$

Take $T_1 = \{\mathit{CurrentStudent}(\mathit{Eve})\}$ and $T_2 = \{\mathit{PastStudent}(\mathit{Eve})\}$. We observe that, both in the 2-valued and the 3-valued settings, the supported-model and the stable model semantics allow to validate T_1 and T_2 (clearly, none of the semantics validate $T_1 \cup T_2$). This might be undesirable in some applications, and here cautious validation becomes handy. Observe that neither T_1 nor T_2 are validated under the cautious semantics. However, naturally $T_3 = \{\mathit{CurrentStudent}(\mathit{Ann}), \mathit{PastStudent}(\mathit{Bob})\}$ does get validated.

Cautions validation can be decided in polynomial time for the case of 3-valued stable model semantics.

Proposition 5.12. *It is a PTIME-complete problem to decide, given an input graph G and a schema (C, T), whether G is valid against (C, T) under cautious 3-valued stable model semantics.*

Proof (Sketch). The upper bound follows from the fact that a schema (C, T) and a graph G always have a *unique* 3-valued stable model I that is maximal w.r.t. shape literals of the form $0.5s(v)$, i.e., that sets as much to "undefined" as possible. This 3-valued stable model is the most "skeptical" among all 3-valued stable models, and can be used to answer cautious validation queries. Importantly, such a representative 3-valued stable model coincides with the well-founded model of the translated program $P_C \cup G_C$ [22], and can therefore be computed in polynomial time [26].

We note that the PTIME lower bound follows from the proof of Proposition 4.10, and holds already for data complexity. □

6 PROTOTYPE IMPLEMENTATION

A key advantage of the stable model semantics for SHACL constraints is that one can outsource the validation problem to an off-the-shelf ASP solver. To test the usefulness of this approach, we have developed a prototype implementation that translates SHACL constraints (in normal form) into ASP programs and evaluated them using the DLV[6] solver. All results were obtained on a MacBook Pro i5 2.7, Sierra OS. Our goal was to measure the runtime of obtaining

[6] http://www.dlvsystem.com/dlv/

Schema	Time(sec) DP_{10}				Time(sec) DP_{20}				Time(sec) DP_{50}			
	2-val	2-val$_{opt}$	3-val	3-val$_{opt}$	2-val	2-val$_{opt}$	3-val	3-val$_{opt}$	2-val	2-val$_{opt}$	3-val	3-val$_{opt}$
S_1	24.22	22.732	25.964	23.624	41.991	39.839	47.645	42.685	91.245	83.235	97.39	90.436
S_2	30.77	28.314	33.592	32.336	62.104	53.612	60.025	59.12	128.698	125.381	146.975	125.381
S_3	20.588	-	24.176	-	20.451	-	41.856	-	95.207	-	98.06	-

Table 5: Evaluation results over DP_{10}, DP_{20} and DP_{50}. Notation n-val$_{opt}$ denotes the optimized version of the ASP program. The translation of S_3 does not use built-in predicates, so no optimization possible.

the translation from input RDF graph G and schema $S = \langle C, T\rangle$ into ASP programs, according to Table 2, and the time needed for the solver to validate the targets w.r.t. the 2-valued and 3-valued stable models semantics. To test for scalability, we used four samples of DPpedia[7] by randomly selecting 10 %, 20 % and 50% of datasets "Person Data", "Instance Types", "Labels", "Mappingbased Literals" and "Mappingbased Objects". Given that non-recursive shemas can be encoded into SPARQL, we have tested 3 recursive schema shapes: S_1 consists of 2 shapes, S_2 of 3 and S_3 of 4. All shapes use constraints of the form $\phi_1 \wedge \phi_2$, $\neg\phi$, and additionally, S_1 and S_2 use also $\geq_n E.\phi$ and $\leq_n E.\phi$, while S_3 uses $\phi_1 \vee \phi_2$. As targets, for all schemas we used singletons of the form $\{s(a)\}$. The input schemas are available online together with the implementation.

The first step in our algorithms was to extract only those triples in the RDF graph that were relevant for the set of constraints. Figure 2 shows that such step can be done reasonably fast. For all schemas, the execution time of the obtained ASP program is less than 1 minute over small datasets such as DP_{10} and DP_{20}, while over DP_{50} is within 2 minutes, which shows that if facts are precomputed, then this approach is feasible in practice. Moreover, we have optimized the ASP encoding by precomputing the grounding of the built-in predicates (i.e., $\mathbf{Next}_E(X, Y, Z)$) w.r.t. the facts from G, and measured the runtime for obtaining a 2-valued, respectively a 3-valued stable model. Table 5 shows the concrete numbers, and particularly, that the optimization leads to a slightly performance boost, especially for larger datasets. We have also counted the number of validated targets per semantics and obtained that for both 2-valued and 3-valued semantics, each dataset over S_1 and S_2 validated the same number of targets. In the case of S_3, for each dataset no 2-valued stable model was obtained, however under the 3-valued semantics we obtained 11, 108, respectively 4066 validated targets.

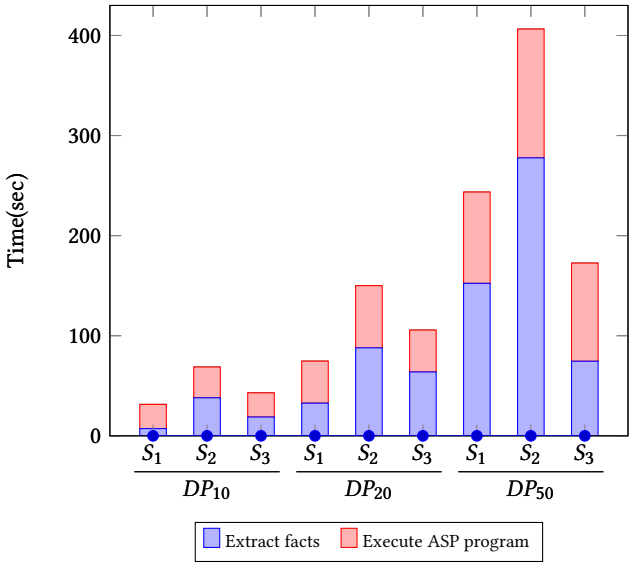

Figure 2: Validation of SHACL schemas using ASP.

7 RELATED WORK

The use of (integrity) constraints in traditional relational databases and in semi-structured data is well established [2, 3]. However, these approaches cannot directly be applied for RDF data, which is graph-structured. When it comes to constraint languages for RDF, it should first be mentioned that RDF Schema (RDFS), contrary to what its name may suggest, is not a schema language in the classical sense, but is primarily used to infer implicit facts. ShEx [8, 24] on the other hand is a proper constraint language for RDF, which shares many similarities with SHACL. In particular, the proposal made in [8] to handle recursive ShEx constraints can be adapted to SHACL. This semantics applies to stratified constraints only: a unique decorated graph is defined by induction on the strata, starting from the lower stratum, and assigning as many shape names as possible, while complying with constraints. An advantage of this approach is the tractability of validation. This semantics is more conservative than supported model semantics, but still more tolerant than stable model semantics. In particular, it may yield unjustified assignments. For instance, it would assign *Elite* to *Ann* in our introductory example.

Prior to ShEx and SHACL, proposals were made to use the Web Ontology Language (OWL), based on Description Logics (DLs) [5], as a constraint language. Like RDFS, OWL was not designed as a schema language, but adopts instead the *open-world assumption*, not well-suited to express constraints. Still, proposals were made to reason with DLs understood as constraints: by introducing *auto-epistemic* operators [12], partitioning DL formulas into regular and constraint axioms [18, 25], or reasoning with closed predicates [21].

The most immediate point of comparison remains the semantics for recursive SHACL proposed in [10], already mentioned. In the stratified case, the 2-valued stable model semantics is strictly more conservative: if a graph is valid against a schema under the 2-valued stable model semantics, then it is also valid under the semantics proposed in [10], but the converse may not hold. This extends to satisfiability: some schemas do not admit any valid graph under the former semantics, but do under the latter. Such schemas are inherently self-supported. This is the case for instance of the schema (C, T) defined by $T = \{EliteShape(Alice)\}$, and

[7] https://wiki.dbpedia.org/develop/datasets/dbpedia-version-2016-10

$C = \{(Elite \leftarrow (\exists hasYacht.\top) \wedge (\exists hasFriend.Elite)\}$. The comparison between 3-valued stable model semantics (defined in Section 5) and the semantics proposed in [10] is less immediate. In particular, both approaches guarantee that a graph and set of constraints must have a unique most "skeptical" model, i.e. a model that intuitively maximizes uncertainty. But interestingly, the most skeptical 3-valued model as defined in [10] and the most skeptical 3-valued stable model do not coincide in general. This suggests that these approaches are not comparable.

8 CONCLUSION

In this paper we have presented and studied a constructive semantics for recursive SHACL, drawing inspiration from the well-understood stable model semantics of logic programs with negation. This allowed us to *i)* establish a principled way to deal with recursion (involving negation) that requires justification for shape assignments, *ii)* create a constructive way of defining semantics for which validation is tractable (in contrast to previous proposals), *iii)* define a guidance on how to implement such semantics using ASP reasoners. Rather than confuse with more semantics, we hope that our work can serve as an input for future discussions on the SHACL standard, and that the question of which is the best semantics is discussed together with input from industry.

In the meantime, we see this work as an important step towards efficient implementation of SHACL reasoners. There is much more work to be done if one wants to lift our prototype implementation to a fully-fledged commercial-grade system, but this is an important and interesting line of work that fosters the connection between the Semantic Web and the logic programming communities.

REFERENCES

[1] Shapes constraint language (SHACL). Technical report, W3C, July 2017.
[2] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.
[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. 1995.
[4] Giovanni Amendola, Thomas Eiter, Michael Fink, Nicola Leone, and JoÃčo Moura. Semi-equilibrium models for paracoherent answer set programs. *Artificial Intelligence*, 234:219 – 271, 2016.
[5] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The description logic handbook: theory, implementation and applications*. Cambridge university press, 2003.
[6] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
[7] Chitta R. Baral and V. S. Subrahmanian. Dualities between alternative semantics for logic programming and nonmonotonic reasoning. *Journal of Automated Reasoning*, 10(3):399–420, Oct 1993.
[8] Iovka Boneva, Jose Emilio Labra Gayo, and Eric G PrudâĂŹhommeaux. Semantics and Validation of Shapes Schemas for RDF. In *ISWC*, 2017.
[9] Julien Corman, Juan L. Reutter, and Ognjen Savkovic. Semantics and validation of recursive SHACL. *ISWC*, 2018.
[10] Julien Corman, Juan L. Reutter, and Ognjen Savkovic. Semantics and validation of recursive SHACL. In *Proc. of ISWC 2018*, volume 11136 of *Lecture Notes in Computer Science*, pages 318–336. Springer, 2018.
[11] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, September 2001.
[12] Francesco M Donini, Daniele Nardi, and Riccardo Rosati. Description logics of minimal knowledge and negation as failure. *ACM Transactions on Computational Logic (TOCL)*, 3(2):177–225, 2002.
[13] T. Eiter, N. Leone, and D. SaccÃă. On the partial semantics for disjunctive deductive databases. *Annals of Mathematics and Artificial Intelligence*, 19(1-2):59–96, 1997. cited By 50.
[14] Fajar J Ekaputra and Xiashuo Lin. SHACL4p: SHACL constraints validation within ProtÃĪgÃĪ ontology editor. In *ICoDSE*, 2016.
[15] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. Keys for graphs. *PVLDB*, 8(12):1590–1601, 2015.
[16] Wenfei Fan, Yinghui Wu, and Jingbo Xu. Functional dependencies for graphs. In *SIGMOD*, pages 1843–1857, 2016.
[17] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.
[18] Boris Motik, Ian Horrocks, and Ulrike Sattler. Bridging the gap between OWL and relational databases. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(2):74–89, 2009.
[19] Mauricio Osorio Galindo, JosÃĪ R. Arrazola RamÃŋrez, and JosÃĪ Luis Carballido. Logical Weak Completions of Paraconsistent Logics. *Journal of Logic and Computation*, 18(6):913–940, 05 2008.
[20] Peter F Patel-Schneider. Using Description Logics for RDF Constraint Checking and Closed-World Recognition. In *AAAI*, 2015.
[21] Peter F Patel-Schneider and Enrico Franconi. Ontology constraints in incomplete and complete data. In *ISWC*, 2012.
[22] Teodor Przymusinski. Well-founded semantics coincides with three-valued stable semantics. *Fundam. Inf.*, 13(4):445–463, October 1990.
[23] Teodor Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9(3-4):401–424, 1991. cited By 231.
[24] Slawek Staworko, Iovka Boneva, Jose Emilio Labra Gayo, Samuel Hym, Eric G Prud'hommeaux, and Harold Solbrig. Complexity and Expressiveness of ShEx for RDF. In *ICDT*, 2015.
[25] Jiao Tao, Evren Sirin, Jie Bao, and Deborah L McGuinness. Integrity Constraints in OWL. In *AAAI*, 2010.
[26] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, July 1991.