

Towards SHACL Validation of Evolving Graphs

Shqiponja Ahmetaj¹, Magdalena Ortiz¹ and Mantas Šimkus¹

¹*Institute of Logic and Computation, TU Wien, Austria*

Abstract

SHACL (SHaPE Constraint Language) is a W3C standardized constraint language for RDF graphs. In this paper, we study SHACL validation in evolving RDF graphs. We identify an update language that can capture intuitive and realistic modifications on RDF graphs and study the problem of static validation under updates, which asks to verify whether a given RDF graph that validates a set of SHACL constraints will remain valid after applying a set of updates. Reducing this problem to usual validation (for a minor SHACL extension) on the current graph allows us to identify problematic updates before applying them to the graph, thus avoiding incorrect and costly computations on potentially huge RDF graphs. More importantly, it provides a basis for further services for reasoning about evolving RDF graphs. In this spirit, we provide preliminary results for a version of *static validation under updates* that verifies whether *every graph* that validates a SHACL specification will still do so after applying a given update sequence. This result builds on previous work that addresses analogous problems but using Description Logics instead of SHACL to describe conditions on graphs.

Keywords

SHACL, static analysis, evolving data graphs, update language

1. Introduction

SHaPE Constraint Language (SHACL), a W3C recommendation since 2017, provides a language for describing conditions on RDF data graphs (for example, objects that belong to the class person can only have one date of birth). In particular, SHACL uses the notion of a *shapes graph* to describe a set of *shape* constraints paired with *targets*, which specify which nodes of the RDF graph should satisfy which shapes. The main computational problem in SHACL is *validation*, which aims to check whether an RDF graph satisfies a shapes graph. SHACL is being increasingly adopted as the basic means to provide quality guarantees on RDF data, and the development of validators and other SHACL tools is progressing quickly. Nevertheless, SHACL is a very recent technology, and many foundational questions remain to be addressed.

RDF graphs can be huge and their validation very costly. When graphs are subjected to updates triggered by users or applications, the validation process may have to be repeated many times. If it turns out that some updates lead to non-validation, returning to the initial valid state may be very difficult, or even impossible. This motivates us to study the effect of updates on SHACL validation, and whenever possible, to reason about such effects leveraging standard technologies like SHACL validators. As a first step, we address the problem of *validation under*

AMW 2024: 16th Alberto Mendelzon International Workshop on Foundations of Data Management, September 30th–October 4th, 2024, Mexico City, Mexico

✉ shqiponja.ahmetaj@tuwien.ac.at (S. Ahmetaj); magdalena.ortiz@tuwien.ac.at (M. Ortiz);
mantas.simkus@tuwien.ac.at (M. Šimkus)

ORCID 0000-0002-NNNNN (S. Ahmetaj); 0000-0002-2344-9658 (M. Ortiz); 0000-0003-0632-0294 (M. Šimkus)



© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

updates, which asks whether a given RDF graph will validate a SHACL shapes graph *after* applying a sequence of updates. In this way, problematic updates may be detected before they are applied, avoiding costly incorrect computations. We also provide preliminary results for a more interesting version of *static validation under updates* that verifies whether *every graph* that validates a SHACL specification will still do so after applying a given update sequence.

In our previous work [1], we explored the dynamics of graph-structured data that evolves through user or application operations, and investigated validation under updates and related reasoning problems. At that time, in the absence of a widely accepted language for describing the states of data instances and expressing constraints on them, we used a custom description logic (DL) [2]—akin to the two-variable fragment of first-order logic with counting quantifiers—that allowed us to effectively reduce static verification to its consistency problem, which is decidable. Now, almost a decade later, SHACL has been developed and standardized specifically for describing the types of constraints and conditions on graphs advocated in that work [1]. Ensuring that SHACL constraints are preserved during RDF graph updates is essential for maintaining data integrity and validity. We revisit the ideas from [1], focusing on SHACL validation during RDF graph updates, and aim to leverage existing SHACL tools to perform validation under these updates.

This work describes our first steps in this direction. We present a framework for specifying transformations on RDF graphs in the presence of SHACL constraints. We propose a language to express modifications on RDF graphs, related to SPARQL Update, but also allowing sequences of updates that admit conditional actions. In particular, the update language uses SHACL shapes for selecting nodes or arcs for modification and for expressing the preconditions in conditional actions. We then study the problem of *validation under updates* in this setting, using SHACL to describe the constraints that are to be preserved and in particular, we adapt the *regression* method from [1], which rewrites the input shapes graph by incorporating the effects of the actions ‘backwards’. This allows us to show that validation under updates can be reduced to standard validation in a small extension of SHACL. We also present some initial work on a stronger, data independent form of *static validation under updates*, which checks whether the execution of a given action preserves the SHACL constraints for every initial data graph. Using the regression technique, we show that static validation under updates can be reduced to (un)satisfiability of a shapes graph in (a minor extension of) SHACL. Since satisfiability is known to be undecidable already for plain SHACL [3], we leverage the results of [4] to identify an expressive fragment for which the problem is feasible in coNEXPTime .

2. SHACL Validation

In this section, we introduce RDF graphs and SHACL *validation*. We follow the abstract syntax and semantics for the fragment of SHACL core studied in [5]; for more details on the W3C specification of SHACL core we refer to [6], and for details of its relation with DLs to [7, 8].

RDF Graphs. We let N_N , N_C , N_P denote countably infinite, mutually disjoint sets of *nodes* (constants), *class names*, and *property names*, respectively. An RDF (data) graph G is a finite set of (ground) *atoms* of the form $B(c)$ and $p(c, d)$, where $B \in N_C$, $p \in N_P$, and $c, d \in N_N$.

Syntax of SHACL. We assume a countably infinite set N_S of *shape names*, disjoint from N_N, N_C, N_P . A *shape atom* is an expression of the form $s(a)$, where $s \in N_S$ and $a \in N_N$. A *path expression* E is a regular expression built using the usual operators $*$, \cdot , \cup from symbols in $N_P^+ = N_P \cup \{p^- \mid p \in N_P\}$, where p^- is the *inverse property* of p . A (complex) *shape* is an expression ϕ obeying the syntax:

$$\phi, \phi' ::= \top \mid s \mid B \mid c \mid \phi \wedge \phi' \mid \neg \phi \mid \geq_n E.\phi \mid E = p \mid \text{disj}(E, p) \mid \text{closed}(P)$$

where $s \in N_S, B \in N_C, c \in N_N, P$ is a set of property names, n is a positive integer, and E is a path expression. In what follows, we write $\phi \vee \phi'$ instead of $\neg(\neg\phi \wedge \neg\phi')$; $\geq_n E$ instead of $\geq_n E.\top$; $\exists E.\phi$ instead of $\geq_1 E.\phi$; $\forall E.\phi$ instead of $\neg\exists E.\neg\phi$.

A (*shape*) *constraint* is an expression of the form $s \leftrightarrow \phi$, where $s \in N_S$ and ϕ is a possibly complex shape. *Targets* in SHACL prescribe that certain nodes of the input data graph should validate certain shapes. A *target expression* is of the form (W, s) , where s is a shape name and W takes one of the following forms:

- constant from N_N , also called *node target*,
- class name from N_C , also called *class target*,
- expressions of the form $\exists p$ with $p \in N_P$, also called *subjects-of target*,
- expressions of the form $\exists p^-$ with $p \in N_P$, also called *objects-of target*.

A target is any set of target expressions. A *shapes graph* is a pair (C, T) , where C is a set of constraints and T is a set of targets. We assume that each shape name appearing in C occurs exactly once on the left-hand side of a constraint, and each shape name occurring in T must also appear in C . A set of constraints C is *recursive*, if there is a shape name in C that directly or indirectly refers to itself. In this work, we focus on *non-recursive* constraints.

Semantics of SHACL. The semantics of SHACL can naturally be given in terms of interpretations. More precisely, an interpretation consists of a non-empty finite domain $\Delta \subset N_N$ and an interpretation function \cdot^I that maps each shape name or class name $Z \in N_S \cup N_C$ to a set $Z^I \subseteq \Delta$ and each property name $p \in N_P$ to a set of pairs $p^I \subseteq \Delta \times \Delta$. The evaluation of complex shape expressions w.r.t. an interpretation I is given in terms of a function \cdot^I that maps a shape expression ϕ to a set of nodes, and a path expression E to a set of pairs of nodes as in Table 1. Then, an interpretation I satisfies a constraint $s \leftrightarrow \phi$ if $s^I = \phi^I$, and I satisfies a shapes graph (C, T) , if I satisfies all constraints in C and $W^I \subseteq s^I$ for each $(W, s) \in T$. If I satisfies Γ , for Γ a constraint, set of constraints, target, or shapes graph, we say I is a model of Γ and denote it in symbols by $I \models \Gamma$. An interpretation I is a *shape assignment* for a data graph G , if $B^I = \{c \mid B(c) \in G\}$ for each class name B , and $p^I = \{(c, d) \mid p(c, d) \in G\}$ for each property name p . Intuitively, I is a shape assignment for G if class and property names are interpreted as specified by G . For simplicity, in this work, we view shape assignments I as sets of atoms of the form $G \cup L$, where $L = \{s(a) \mid a \in s^I\}$ is a set of shape atoms. A data graph G *validates* a shapes graph (C, T) if there exists a shape assignment I for G that satisfies (C, T) .

Clearly, for non-recursive constraints, which is the setting we consider here, the unique assignment $I_{G,C} = G \cup L_{G,C}$ obtained in a bottom-up fashion, starting from G and evaluating each constraint once, is a model of C . More precisely, since the constraints in C are non-recursive,

$$\begin{aligned}
\top^I &= \Delta & c^I &= \{c\} & A^I &= \{c \mid A(c) \in I\} \\
s^I &= \{c \mid s(c) \in I\} & (\neg\phi)^I &= \Delta \setminus (\phi)^I & (\phi_1 \wedge \phi_2)^I &= (\phi_1)^I \cap (\phi_2)^I \\
(\geq_n E.\phi)^I &= \{c \mid |\{(c, d) \in (E)^I \text{ and } d \in (\phi)^I\}| \geq n\} \\
(E = p)^I &= \{c \mid \forall d : (c, d) \in (E)^I \text{ iff } (c, d) \in (p)^I\} \\
(\text{disj}(E, p))^I &= \{c \mid \exists d : (c, d) \in (E)^I \text{ and } (c, d) \in (p)^I\} \\
(\text{closed}(P))^I &= \{c \mid \forall p \notin P : c \notin (\exists p)^I\} \\
(p)^I &= \{(a, b) \mid p(a, b) \in I\} & (p^-)^I &= \{(a, b) \mid p(b, a) \in I\} \\
(E \cdot E')^I &= \{(a, b) \mid \exists d : (a, d) \in (E)^I \text{ and } (d, b) \in (E')^I\} \\
(E \cup E')^I &= (E)^I \cup (E')^I & (E^*)^I &= \{(a, a) \mid a \in \Delta\} \cup (E)^I \cup (E \cdot E)^I \cup \dots
\end{aligned}$$

Table 1
Semantics of SHACL shape expressions

one can start from constraints of the form $s_1 \leftrightarrow \phi_1$, where ϕ_1 has no shape names, and add all the atoms $s(a)$ such that $a \in \llbracket \phi_1 \rrbracket^G$ and obtain $G \cup L_1$. We then proceed with constraints of the form $s_2 \leftrightarrow \phi_2$, where ϕ_2 has only shape names occurring in L_1 and add in L_2 all $s_2(a)$ such that $a \in \llbracket \phi_2 \rrbracket^{G \cup L_1}$; the new assignment is $G \cup L_1 \cup L_2$. We proceed like this with the rest of the constraints, by evaluating each of them once, until all of them are processed. The resulting assignment $I_{G,C}$ is unique and is a model that satisfies all the constraints in C ; we call $I_{G,C} = G \cup L_{G,C}$ the model of G and C . Then, we can formulate validation as follows: a data graph G *validates* (C, T) if $I_{G,C}$ satisfies (W, s) for every target $(W, s) \in T$.

Example 2.1. Consider the following data graph G and shapes graph (C, T) :

$$\begin{aligned}
G &= \{\text{Prj}(p_1), \text{Prj}(p_2), \text{ActivePrj}(p_1), \text{ActivePrj}(p_2), \text{Empl}(\text{Ann}), \text{Empl}(\text{Ben}), \text{Empl}(\text{Tom}), \\
&\quad \text{worksFor}(\text{Ann}, p_1), \text{worksFor}(\text{Ben}, p_1), \text{worksFor}(\text{Tom}, p_2)\} \\
C &= \{\text{PrjShape} \leftrightarrow \text{ActivePrj} \vee \text{FinishedPrj}, \\
&\quad \text{EmplShape} \leftrightarrow \text{Empl} \vee \exists \text{worksFor}.\text{ActivePrj}\}, \\
T &= \{(\text{Prj}, \text{PrjShape}), (\exists \text{worksFor}, \text{EmplShape})\}
\end{aligned}$$

Intuitively, the data graph contains 2 projects, p_1, p_2 which are also active projects, as well as three employees, Ann, Ben, Tom; the first two work for p_1 and Tom works for p_2 . Together, the constraints and target intuitively state that all projects must be either active or finished, and all those working for some projects must be instances of the class Empl or work for some active project. In this case, G validates (C, T) ; this is witnessed by the assignment $G \cup \{\text{PrjShape}(p_1), \text{PrjShape}(p_2), \text{EmplShape}(\text{Ann}), \text{EmplShape}(\text{Ben}), \text{EmplShape}(\text{Tom})\}$ which satisfies T .

3. Extending SHACL for Evolving RDF Graphs

In this section, we propose an extension of SHACL with a construct (a, b) for a singleton property and with difference of properties. We also allow for complex target expressions (W, s) , where W

can now be any complex shape expression without shape names, and we allow for variables to occur in the place of constants. Moreover, we allow for targets to support Boolean combinations of target expressions. We denote with SHACL^+ this extended version of SHACL. This extension turns out to be necessary to capture the effects of actions on SHACL constraints and to express preconditions in the update language defined in the following section.

Assume a countable set N_V of variables (disjoint from the rest). More precisely, SHACL^+ properties are defined according to the following syntax:

$$r, r' ::= p \mid (t_1, t_2) \mid r^- \mid r \setminus r'$$

where $t_1, t_2 \in N_I \cup N_V$ and $p \in N_P$; we call them *complex* properties. Paths E over complex properties and (*complex*) shape expressions over such paths are defined as expected. Targets are extended to allow variables in the places of nodes and complex shapes and Boolean combinations of target expressions as well.

More formally, SHACL^+ *target expressions* are defined as follows:

- (W, s) , where W is a complex shape expression without shape names, is a target,
- a target expression (t, s) , where $t \in N_V \cup N_I$ is a target,
- if T_1 and T_2 are targets, then $T_1 \wedge T_2$, $T_1 \vee T_2$, and $\neg T_1$ are targets.

If no variables appear in C (or T), then it is called *ground*. Now, when an assignment is a model of a target is naturally defined as follows.

Definition 3.1. Consider a shape assignment I and a ground target in SHACL^+ . Then, I is a model of T , that is $I \models T$, if the following hold:

- $W^I \in s^I$, if T is a target expression of the form (W, s) ,
- $I \models T_1$ and $I \models T_2$, if T is of the form $T_1 \wedge T_2$,
- $I \models T_1$ or $I \models T_2$, if T is of the form $T_1 \vee T_2$, and
- $I \not\models T_1$, if T is of the form $\neg T_1$

Finally, in SHACL^+ we also allow for Boolean combinations of shapes graphs. More formally, a SHACL^+ *shapes graph* is defined recursively as follows:

- (C, T) is a SHACL^+ shapes graph, where C and T are ground SHACL^+ sets of constraints and targets, respectively, and
- if $\mathcal{S}_1, \mathcal{S}_2$ are SHACL^+ shapes graphs, then $\mathcal{S}_1 \wedge \mathcal{S}_2$, $\mathcal{S}_1 \vee \mathcal{S}_2$, and $\neg \mathcal{S}_1$ are SHACL^+ shapes graphs.

A SHACL^+ shapes graph is called *normal*, if it is of the form (C, T) , where C and T are ground SHACL^+ constraints and targets, respectively. Validation is naturally defined as follows.

Definition 3.2. Consider SHACL^+ shapes graph \mathcal{S} and data graph G . Then, G validates \mathcal{S} , if the following hold:

- $I_{G,C} \models T$, if \mathcal{S} is of the form (C, T) ,
- G validates \mathcal{S}_1 and \mathcal{S}_2 if \mathcal{S} is of the form $\mathcal{S}_1 \wedge \mathcal{S}_2$,
- G validates \mathcal{S}_1 or \mathcal{S}_2 if \mathcal{S} is of the form $\mathcal{S}_1 \vee \mathcal{S}_2$, and
- G does not validate \mathcal{S}_1 if \mathcal{S} is of the form $\neg \mathcal{S}_1$

Allowing Boolean combinations of shapes graphs is just syntactic sugar, as each SHACL^+ shapes graphs \mathcal{S} can be converted into equivalent normal shapes graphs by simply renaming the

shape names in each normal shapes graph appearing in \mathcal{S} , taking the union of all constraints, and pushing the Boolean operators to the targets.

Proposition 3.3. *Consider a shapes graph \mathcal{S} in SHACL^+ and consider a data graph G . Then, \mathcal{S} can be converted in linear time into a normal SHACL^+ shapes graph $(C_{\mathcal{S}}, T_{\mathcal{S}})$ such that for every data graph G , it holds that G validates \mathcal{S} iff G validates $(C_{\mathcal{S}}, T_{\mathcal{S}})$.*

The above proposition shows that allowing Boolean combinations of shapes graphs is just syntactic sugar, since we can equivalently transform such shapes graphs into a unique normal shapes graph (C, T) in SHACL^+ .

4. Update Language for RDF Graphs

We now present the action language for updating RDF graphs. It is composed of two types of actions, namely basic and complex actions. Roughly, the basic actions allow to insert or delete constants from the evaluation of a shape expression or arbitrary property over the graph to a class or property name, respectively. The complex actions allow for composing various actions and to perform conditional checks over the data. In particular, we assume that a set of SHACL constraints that contains all the necessary and sufficient shape constraints is given in the input together with the set of actions. The preconditions in the actions are targets in SHACL^+ , that is any Boolean combination of target expressions that will be checked w.r.t. a set of constraints.

Basic actions β are defined by the following grammar:

$$\beta ::= (A \xleftarrow{\oplus} \phi) \mid (A \xleftarrow{\ominus} \phi) \mid (p \xleftarrow{\oplus} r) \mid (p \xleftarrow{\ominus} r)$$

where A is a class name, ϕ a complex shape without shape names, p a property name, and r a complex property. *Complex actions* α are defined by the following grammar:

$$\alpha ::= \emptyset \mid \beta \cdot \alpha \mid ((C, T^+)?\alpha[\alpha]) \cdot \alpha$$

where β is a basic action, \emptyset is an empty action, α is a complex action, and (C, T^+) is a SHACL^+ normal shapes graph. We may sometimes write $(T^+?\alpha[\alpha]) \cdot \alpha$, omitting C from the actions if C is given in the input, and the precondition targets T^+ are required to be applied on the input C .

A *substitution* is a function σ from N_V to N_N . For a Boolean target, an action, or an action sequence γ , we use $\sigma(\gamma)$ to denote the result of replacing in γ every occurrence of a variable x by a constant $\sigma(x)$. An action α is ground if it has no variables, and α' is a ground instance of an action α if $\alpha' = \sigma(\alpha)$ for some substitution σ .

Intuitively, an application of a ground action $(A \xleftarrow{\oplus} \phi)$ on a graph G and a ground set of constraints C , stands for the addition of $A(c)$ to G for each c that makes true ϕ in the model $I_{G,C}$ of G and C ; analogously for $(A \xleftarrow{\ominus} \phi)$, where now $A(c)$ will be removed from G for each c that makes true ϕ in $I_{G,C}$. The two operations can also be performed on complex properties. Composition stands for successive action execution, and a conditional action $((C, T^+)?\alpha_1[\alpha_2])$ expresses that α_1 is executed if T is true in $I_{G,C}$, and α_2 is performed otherwise. If $\alpha_2 = \emptyset$, then we have an action with a simple precondition as in classical planning languages, and we write

it as $(C, T^+)?\alpha_1$, without α_2 . The semantics of applying actions on data graphs, w.r.t. a set of constraints, is defined only on ground actions and constraints.

Definition 4.1. *Let G be a data graph, and α a ground complex action. Then, the result G^α of applying α on G is defined recursively as follows:*

- if α is a basic action, then
 - $G^\alpha = G \cup \{A(a) \mid a \in \phi^I\}$ for α of the form $(A \xleftarrow{\oplus} \phi)$,
 - $G^\alpha = G \setminus \{A(a) \mid a \in \phi^I\}$ for α of the form $(A \xleftarrow{\ominus} \phi)$,
 - $G^\alpha = G \cup \{p(a, b) \mid (a, b) \in r^I\}$ for α of the form $(p \xleftarrow{\oplus} r)$, and
 - $G^\alpha = G \setminus \{p(a, b) \mid (a, b) \in r^I\}$ for α of the form $(p \xleftarrow{\ominus} r)$
- if α is a complex action γ of the form $\beta \cdot \alpha$, then $G^\gamma = (G^\beta)^\alpha$
- if α is a complex action γ of the form $((C, T^+)?\alpha_1[\alpha_2]) \cdot \alpha$, then G^γ is
 - $G^{\alpha_1 \cdot \alpha}$, if $I_{G, C}$ satisfies T ,
 - $G^{\alpha_2 \cdot \alpha}$ if $I_{G, C}$ does not satisfy T .

We illustrate the effects of an action update on a data graph with an example.

Example 4.2. *Consider again G and (C, T) from Example 2.1. Now, consider the action α that expresses the termination of project p_2 , which is removed from the active projects and added to the finished ones; the employees working only for this project are removed.*

$$\alpha = (\text{ActivePrj} \xleftarrow{\ominus} p_2) \cdot (\text{FinishedPrj} \xleftarrow{\oplus} p_2) \cdot (\text{Empl} \xleftarrow{\ominus} \forall \text{worksFor}. p_2)$$

After applying α to G , we obtain the updated data graph $G^\alpha = (G \setminus \{\text{ActivePrj}(p_2), \text{Empl}(\text{Tom})\}) \cup \{\text{FinishedPrj}(p_2)\}$. The updated data graph G^α does not validate the shapes graph (C, T) since now Tom will still have a worksFor -relation to project p_2 , but he does not satisfy the constraint for the shape name EmplShape — Tom is not an employee and does not work for an active project.

Note that we have not defined the semantics of actions with variables, that is non-ground actions. In our approach, all variables of an action are seen as parameters whose values are given before execution by a substitution with actual constants, such as by grounding. Note that the execution of actions on an initial data graph is allowed to modify the extensions of class and property names, yet the domain remains fixed. In many scenarios, we would like actions to have the ability to introduce “fresh” nodes to a data graph. Intuitively, the introduction of new nodes can be modeled in our setting by separating the domain of an assignment into the active domain and the inactive domain. The active domain consists of all nodes that occur in the data graph, whereas the inactive domain contains the remaining nodes, which can be seen as a supply of fresh nodes that can be introduced into the active domain by executing actions. Since we are interested only in finite sequences of actions, a sufficient supply of fresh constants can always be ensured by taking a sufficiently large yet still finite inactive domain in the initial instance. We remark also that deletion of nodes can naturally be modeled in this setting by actions that move objects from the active domain to the inactive domain.

Example 4.3. Consider again our running example. Now consider the following action:

$$\alpha' = (\{s \leftrightarrow \forall \text{worksFor}.p_2\}, \{(t, s)\})?(\text{worksFor} \xleftarrow{\ominus} \{(t, p_2)\})$$

Under the substitution $\sigma(t) = \text{Tom}$, which maps the variable t to Tom , the data graph G validates the shapes graph $(\{s \leftrightarrow \forall \text{worksFor}.p_2\}, \{(\text{Tom}, s)\})$. Hence, the ground action α'_{Tom} can be applied to G^α by deleting $\text{worksFor}(\text{Tom}, p_2)$. The resulting data graph satisfies (C, T) . Thus, the application of the ground complex action $\alpha \cdot \alpha'_{\text{Tom}}$ on G results in a valid data graph.

5. Capturing Effects of Updates

In this section, we define a transformation $TR_\alpha(C, T)$ that essentially rewrites the input shapes graph (C, T) to capture all the effects of an action α . This transformation can be seen as a form of regression, which incorporates the effects of a sequence of actions “backwards,” from the last one to the first one. More precisely, the transformation $TR_\alpha(C, T)$ takes a SHACL shapes graph (C, T) and an action α and rewrites them into a new shapes graph (C_α, T_α) , possibly in SHACL⁺, such that for any data graph G the following holds:

$$G^\alpha \text{ validates } (C, T) \text{ iff } G \text{ validates } (C_\alpha, T_\alpha)$$

If α is without preconditions, the result of applying the transformation is a shapes graph (C^α, T^α) , where C^α is a SHACL set of constraints, and T^α is a set of target expressions (W, s) (similarly as in plain SHACL), but may contain complex shape expressions in place of W . Intuitively, the idea is to simply update the bodies of constraints in C and target expressions in T accordingly, namely the actions replace a concept name A occurring in C and T with $A \wedge C$ (or $A \wedge \neg C$) if the action is $A \xleftarrow{\oplus} C$ (or $A \xleftarrow{\ominus} C$), and a property p with $p \cup r$ (or $p \setminus r$) if the action is $p \xleftarrow{\oplus} r$ (or $p \xleftarrow{\ominus} r$). Now consider the case, where the action is of the form $(C, T^+)?\alpha_1[\alpha_2]$. Then, intuitively, we create two shapes graphs: one shapes graph $(C_{\alpha_1}, T_{\alpha_1})$ if the SHACL⁺ target T^+ is satisfied by $I_{C,G}$ and $(C_{\alpha_2}, T_{\alpha_2})$ if the SHACL⁺ target T^+ is not satisfied by $I_{C,G}$.

Definition 5.1. Assume SHACL⁺ shapes graph \mathcal{S} and an action α . We use $\mathcal{S}_{E \leftarrow E'}$ to denote the new SHACL⁺ shapes graph that is obtained from \mathcal{S} by replacing in \mathcal{S} every class or property name E with the expression E' . Then, the transformation $TR_\alpha(\mathcal{S})$ of \mathcal{S} w.r.t., α is defined recursively as follows:

$$\begin{aligned} TR_\epsilon(\mathcal{S}) &= \mathcal{S} \\ TR_{(A \xleftarrow{\oplus} C) \cdot \alpha}(\mathcal{S}) &= (TR_\alpha(\mathcal{S}))_{A \leftarrow A \vee C} \\ TR_{(A \xleftarrow{\ominus} C) \cdot \alpha}(\mathcal{S}) &= (TR_\alpha(\mathcal{S}))_{A \leftarrow A \wedge \neg C} \\ TR_{(p \xleftarrow{\oplus} r) \cdot \alpha}(\mathcal{S}) &= (TR_\alpha(\mathcal{S}))_{p \leftarrow p \cup r} \\ TR_{(p \xleftarrow{\ominus} r) \cdot \alpha}(\mathcal{S}) &= (TR_\alpha(\mathcal{S}))_{p \leftarrow p \setminus r} \\ TR_{((C, T^+)?\alpha_1[\alpha_2]) \cdot \alpha}(\mathcal{S}) &= (\neg(C, T^+) \vee TR_{\alpha_1 \cdot \alpha}(\mathcal{S})) \wedge ((C, T^+) \vee TR_{\alpha_2 \cdot \alpha}(\mathcal{S})) \end{aligned}$$

Note that in the presence of conditional actions, we may obtain Boolean combinations of normal shapes graphs, whose number may be exponential in the size of the input action α . However, each of the normal shapes graphs will be of polynomial size. When no conditional actions are present, then the result of the transformation on a SHACL shapes graph is a normal SHACL⁺ shapes graph that simply uses role difference in constraints (role union can be captured by paths) and possibly complex expressions without shape names in target expressions. Moreover, note that by proceeding as in Proposition 3.3, the result of $TR_\alpha(\mathcal{S})$ can be converted in an equivalent SHACL⁺ normal shapes graph (C_α, T_α) .

Using similar arguments as in the proof of Theorem 4.2 in [1], and arguing about the additional constructs allowed in SHACL⁺, we can show that this transformation correctly captures the effects of complex actions.

Theorem 5.2. *Given a ground action α , a data graph G and a SHACL shapes graph \mathcal{S} . Then, G^α validates \mathcal{S} if and only if G validates $TR_\alpha(\mathcal{S})$.*

This allows us to perform validation under updates leveraging standard, static validators for SHACL⁺. We illustrate the transformation above with an example.

Example 5.3. *Consider the shapes graph (C, T) and the action α from our running example. Then, the transformation $TR_\alpha((C, T))$ of (C, T) w.r.t. α is the new shapes graph (C', T') , where:*

$$\begin{aligned} C' &= \{ \text{PrjShape} \leftrightarrow (\text{ActivePrj} \wedge \neg p_2) \vee (\text{FinishedPrj} \vee p_2), \\ &\quad \text{EmplShape} \leftrightarrow (\text{Empl} \wedge \neg \forall \text{worksFor}. p_2) \vee \exists \text{worksFor}. (\text{ActivePrj} \wedge \neg p_2) \}, \\ T' &= \{ (\text{Prj}, \text{PrjShape}), (\exists \text{worksFor}, \text{EmplShape}) \} \end{aligned}$$

6. Static Validation under Updates for Arbitrary Graphs

In this section, we consider a stronger form of reasoning about updates. We have seen that, for each input graph, validation under updates can be reduced to usual validation. Now we look at whether a sequence of actions is compatible with the constraints, independently of a concrete data graph, that is, whether the execution of a given sequence of actions always preserves the satisfaction of a given shapes graph, for every data graph.

Definition 6.1. *Let \mathcal{S} be a shapes graph and let α be an action. Then α is a \mathcal{S} -preserving action if for every data graph G that validates \mathcal{S} , it holds that G^α validates \mathcal{S} . The static validation under updates problem is:*

Given an action α and a shapes graph \mathcal{S} , is α \mathcal{S} -preserving?

Using the transformation from Definition 5.1, we can reduce static validation under updates to unsatisfiability of shapes graphs: an action α is not \mathcal{S} -preserving if and only if there is some data graph that does not validate $TR_{\alpha^*}(\mathcal{S})$, where α^* is obtained from α by replacing each variable with a fresh constant name not occurring in the input.

Analogously to Theorem 5.2 in [1], we can show the correctness of the following theorem.

Theorem 6.2. *Let α be a complex action and \mathcal{S} a SHACL⁺ shapes graph. The following are equivalent:*

- α is not \mathcal{S} -preserving,
- $\mathcal{S} \wedge \neg TR_{\alpha^*}(\mathcal{S})$ is satisfiable, where α^* is obtained from α by replacing each variable with a fresh constant not occurring in \mathcal{S} and α .

It has been shown in [3] that checking satisfiability is undecidable already for (plain) SHACL shapes graphs. However, if we restricts the constructs allowed in the $SHACL^+$ shape expressions to the $ALCHOIQ^{br}$ fragment studied in [1], we can obtain better upper-bounds. In particular, it was shown in [1] that the static validation under updates problem is in $coNEXPTIME$ when the input is in $ALCHOIQ^{br}$ and in $EXPTIME$ when the input KB is expressed in $ALCHOI$. Hence, if we disallow path expressions (other than role union, which can be allowed), equality, disjointness, and closed constraints in $SHACL^+$ shapes graphs, then the resulting $\mathcal{S} \wedge \neg TR_{\alpha^*}(\mathcal{S})$ can be easily converted into an $ALCHOIQ^{br}$ knowledge base—with shape names, which can be treated as class names—showing the membership in $coNEXPTIME$. Further, if we restrict cardinality constraints and are left with $ALCHOI$ knowledge bases, then we obtain membership in $EXPTIME$. Note that using similar arguments as in [1] we can argue that each normal shapes graph appearing in $\mathcal{S} \wedge \neg TR_{\alpha^*}(\mathcal{S})$ is of polynomial size. For the lower bounds, we can reduce the $NEXPTIME$ problem of finite satisfiability of $ALCHOIQ$ (or $EXPTIME$ of $ALCHOI$) into the co-problem of static validation for SHACL under updates.

7. Conclusion and Future Work

We have presented some preliminary work on formalizing updates for RDF graphs over SHACL constraints. We addressed an important question: validation under updates, which asks to verify whether a given RDF graph that validates a set of SHACL constraints, will remain valid after applying a set of updates. This allows to identify problematic updates before applying them to the graph, and thus to avoid incorrect and very costly computations on potentially huge RDF graphs. In particular, it is beneficial to know if a set of updates leads to a valid graph in case returning to the initial valid state is difficult, or that the user does not have sufficient permissions in the system. To realize this form of static validation, we first identify a suitable update language, that can capture intuitive and realistic modifications on RDF graphs, covers a significant fragment of SPARQL updates and extends them to allow for conditional updates.

We also present a stronger form of static validation under updates that considers validation on every graph that validates a given SHACL shapes graph. We show that the latter problem can be reduced to (un)satisfiability of a shapes graph in $SHACL^+$, a minor extension of SHACL, which is known to be undecidable already for plain SHACL, but feasible in $coNEXPTIME$ for some expressive fragments.

$SHACL^+$ mainly extends SHACL with singleton properties and difference of properties, which are needed to capture the effects of the proposed actions on SHACL constraints. We also allow for more complex targets. However, most of the extensions allowed in the targets in this work are “syntactic sugar” and can be incorporated into plain SHACL. We note that the SHACL Community Group is working on extending SHACL to support some of the features allowed in this work, including more expressive targets as so-called SPARQL-based targets, with ongoing

review and documentation in the SHACL Advanced Features draft¹.

Toward lowering the complexity of static validation, we plan to analyze the problem for other relevant fragments of SHACL. We will also study other basic static analysis problems such as, for instance, the static type checking problem [9], which for a given action, a source and target shapes graph, asks whether, for every data graph that validates the source shapes graph, the data graph after applying the action also validates the target shapes graph. Other interesting problems related to planning ask to check whether there exists a sequence of actions that leads a given data graph into a desired (or undesired) state where some property holds (or does not hold). An important direction for future work is to provide an implementation of the regression method. This method allows us to perform validation under updates by leveraging standard validators, provided that they can support SHACL⁺; this is not the case for current validators, but we believe that this extension is not hard to incorporate to existing validators, and plan to address this in our future work.

Acknowledgements

This work was supported by the Austrian Science Fund (FWF) and netidee SCIENCE project T1349-N and the FWF project P30873.

References

- [1] S. Ahmetaj, D. Calvanese, M. Ortiz, M. Simkus, Managing change in graph-structured data using description logics, *ACM Trans. Comput. Log.* 18 (2017) 27:1–27:35. URL: <https://doi.org/10.1145/3143803>. doi:10.1145/3143803.
- [2] F. Baader, I. Horrocks, C. Lutz, U. Sattler, *An Introduction to Description Logic*, Cambridge University Press, 2017. URL: <http://www.cambridge.org/de/academic/subjects/computer-science/knowledge-management-databases-and-data-mining/introduction-description-logic?format=PB#17zVGeWD2TZUeu6s.97>.
- [3] P. Pareti, G. Konstantinidis, F. Mogavero, T. J. Norman, SHACL satisfiability and containment, in: *The Semantic Web - ISWC 2020 - 19th International Semantic Web Conference*, volume 12506 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 474–493. URL: https://doi.org/10.1007/978-3-030-62419-4_27. doi:10.1007/978-3-030-62419-4_27.
- [4] S. Ahmetaj, D. Calvanese, M. Ortiz, M. Šimkus, Managing change in graph-structured data using description logics, *ACM Trans. Comput. Log.* (2017). URL: <https://doi.org/10.1145/3143803>. doi:10.1145/3143803.
- [5] S. Ahmetaj, R. David, M. Ortiz, A. Polleres, B. Shehu, M. Simkus, Reasoning about explanations for non-validation in SHACL, in: *Proceedings of KR 2021, Online event, November 3-12, 2021, 2021*, pp. 12–21. URL: <https://doi.org/10.24963/kr.2021/2>. doi:10.24963/kr.2021/2.
- [6] Shapes constraint language (SHACL), 2017. URL: <https://www.w3.org/TR/shacl/>.
- [7] M. Ortiz, A short introduction to SHACL for logicians, in: *Logic, Language, Information, and Computation - 29th International Workshop, WoLLIC 2023*, volume 13923 of

¹<https://w3c.github.io/shacl/shacl-af/>

Lecture Notes in Computer Science, Springer, 2023, pp. 19–32. URL: https://doi.org/10.1007/978-3-031-39784-4_2. doi:10.1007/978-3-031-39784-4_2.

- [8] B. Bogaerts, M. Jakubowski, J. V. den Bussche, SHACL: A description logic in disguise, in: G. Gottlob, D. Incezan, M. Maratea (Eds.), *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR*, volume 13416 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 75–88. URL: https://doi.org/10.1007/978-3-031-15707-3_7. doi:10.1007/978-3-031-15707-3_7.
- [9] I. Boneva, B. Groz, J. Hidders, F. Murlak, S. Staworko, Static analysis of graph database transformations, in: *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS*, ACM, 2023, pp. 251–261. URL: <https://doi.org/10.1145/3584372.3588654>. doi:10.1145/3584372.3588654.