Repairing SHACL Constraint Violations using Answer Set Programming

Shqiponja Ahmetaj[1,2], Robert David[2,4], Axel Polleres[2,3], and Mantas Šimkus[1,5]

[1]TU Wien, [2] WU Wien, [3]Complexity Science Hub Vienna, [4]Semantic Web Company, [5]Umeå University

`shqiponja.ahmetaj@tuwien.ac.at, robert.david@semantic-web.com, axel.polleres@wu.ac.at, simkus@cs.umu.se`

Abstract. The Shapes Constraint Language (SHACL) is a recent W3C recommendation for validating RDF graphs against *shape* constraints to be checked on *target nodes* of the data graph. The standard also describes the notion of *validation reports* for data graphs that violate given constraints, which aims to provide feedback on how the data graph can be fixed to satisfy the constraints. Since the specification left it open to SHACL processors to define such explanations, a recent work proposed the use of explanations in the style of database *repairs*, where a repair is a set of additions to or deletions from the data graph so that the resulting graph validates against the constraints. In this paper, we study such repairs for non-recursive SHACL, the largest fragment of SHACL that is fully defined in the specification. We propose an algorithm to compute repairs by encoding the explanation problem – using Answer Set Programming (ASP) – into a logic program, the answer sets of which correspond to (minimal) repairs. We then study a scenario where it is not possible to simultaneously repair all the targets, which may be often the case due to overall unsatisfiability or conflicting constraints. We introduce a relaxed notion of validation, which allows to validate a (maximal) subset of the targets and adapt the ASP translation to take into account this relaxation. Our implementation in Clingo is – to the best of our knowledge – the first implementation of a repair generator for SHACL.

Keywords: SHACL · Shapes Constraint Language · Database Repairs · RDF Graphs · Semantic Web.

1 Introduction

Semantic Web standards provide means to represent and link heterogeneous data sources in knowledge graphs [10], thereby potentially solving common data integration problems. Indeed, this approach became increasingly popular in enterprises for the consolidation of data silos in the form of so-called *enterprise knowledge graphs (EKG)*. However, in practice this flexible and expressive approach to data integration requires powerful tools for ensuring data quality, including ways to avoid creating invalid data and inconsistencies in the target

EKGs. To this end, the W3C proposed the Shapes Constraint Language SHACL, in order to enable validation of RDF graphs against a set of shape constraints [1]. In this setting, the validation requirements are specified in a *shapes graph* (C, T) that consists of a collection C of validation rules (constraints) and a specification T of nodes to which various constraints should be applied. The result of validating an RDF graph (or, *data graph*) G against a shapes graph (C, T) is a *validation report*, which lists the constraint violations present in G. Unfortunately, validation reports, as specified in the SHACL standard, contain little information on what steps could be made to resolve those constraint violations. Since in many common scenarios (like the automated integration of heterogeneous data sources) inconsistencies appear very frequently, there is a need to *automatically* identify repairs that can be applied to the data graph in order to achieve consistency. A *repair* in our context is a collection of additions and deletions of facts that will cause the data to be consistent with the given constraints. Our contributions are as follows:

◦ We propose to compute repairs of a data graph by encoding the problem into *Answer Set Programming (ASP)* [7]. In particular, we show how to transform a given data graph G and a SHACL shapes graph (C, T) into an ASP program P such that the answer sets (or, stable models) of P can be seen as a collection of plausible repairs of G w.r.t. the shapes graph (C, T). Since efficient ASP solvers exist (we use Clingo [9]), this provides a promising way to generate data repairs in practice. The repair generation task is challenging, because a given data graph might be repaired in many different ways. In fact, since fresh nodes could be introduced during the repair process, an infinite number of repairs is possible. This needs to be handled carefully and several design choices are possible.

◦ We initially present the *basic encoding* of the repair task into ASP. In this encoding, the repair program tries to find a repair that satisfies *all* targets of the input shapes graph. This encoding employs a particular strategy for introducing new nodes in the data graph: when a value for a property needs to be added (e.g. for a violated *sh*:*minCount* constraint), a fresh value is always introduced. We argue that it is a reasonable strategy; it is also closely related to the standard notion of *Skolemization*. By using some of the features of ASP, we ensure that our repair program generates repairs that are minimal in terms of cardinality, which means that they contain only minimal modifications for resolving constraint violations. Our basic encoding is later extended to allow for the introduction of fresh nodes as well as the reuse of existing or previously introduced nodes.

◦ We observe that requiring a repair to resolve violations for *all* specified targets may be too strong. In the context of the basic encoding, if the data graph has one inherently unfixable target (e.g., because of some erroneous constraint), then the repair program will have no answer sets at all and it will provide no guidance on how to proceed with fixing the data graph. To address this issue, we introduce the notion of *maximal repairs*, which repair the highest number of targets that is possible to repair. We show how our encoding can be augmented to generate repairs according to this new notion. This is done using the optimization features of Clingo as well as rules that allow to skip some targets.

∘ We have implemented and tested these encodings using the Clingo ASP system, which showed that our approach is promising for providing quality control and quality improvements for RDF graphs for practical use.

Related Work. Our approach is inspired by previous work in the area of databases on computing repairs for violations of database integrity constraints (see, e.g., [5]) and reasoning about them. We adapt it for the RDF data model and SHACL constraints. Close to our work is [11], where database repairs are specified using disjunctive logic programs with the answer set semantics. These repairs modify a database to achieve conformance with a set of integrity constraints that are applied to a relational database. The repair program uses *annotations* to indicate which atoms should be added or deleted to satisfy the constraints. The program contains rules whose body identifies a violation of a constraint, while a disjunctive rule head describes the candidate actions (additions and deletions of tuples) that can potentially be used to resolve the identified violation. These repair rules can interact and possibly resolve conflicting constraints, eventually stabilizing into a minimal repair. The repair program contains constraints to prevent models with conflicting insertions and deletions. The so-called *interpretation* rules are then used to collect the actual additions and deletions corresponding to a possible database repair.

2 SHACL Validation and Answer Set Programming

In this section, we describe SHACL [1] and the notion of *validation* against RDF graphs. For an introduction to data validation, SHACL, and its close relative ShEx, we refer to [8]. We also describe answer set programming (ASP), which we use to implement the repair program.

SHACL Validation We use the abstract syntax from [2] for RDF and SHACL. Note that in this work we focus on the fragment of SHACL 'Core Constraint Components' without path expressions (except for inverse roles), equality and disjoint operators.

Data graph. We first define *data graphs*[1], which are RDF graphs to be validated against shape constraints. Assume countably infinite, mutually disjoint sets $\mathbf{N}$, $\mathbf{C}$, and $\mathbf{P}$ of *nodes* (or *constants*), *class names*, and *property names*, respectively. A *data graph* G is a finite set of *(ground) RDF atoms* of the form $B(c)$ and $p(c, d)$, where B is a class name, p is a property name, and c, d are nodes.

Syntax of SHACL. Let $\mathbf{S}$ be a countably infinite set of *shape names*, disjoint from $\mathbf{N}$, $\mathbf{C}$, and $\mathbf{P}$. A *shape expression* ϕ is of the form:

$$\phi, \phi' ::= \top \mid s \mid B \mid c \mid \phi \wedge \phi' \mid \neg\phi \mid \geq_n r.\phi \tag{1}$$

where $s \in \mathbf{S}$, $B \in \mathbf{C}$, $c \in \mathbf{N}$, n is a positive integer, and r is a property $p \in \mathbf{P}$ or an *inverse property* of the form p^- with $p \in \mathbf{P}$. In what follows, we may write

[1] `https://www.w3.org/TR/shacl/#data-graph`

$\phi \vee \phi'$ instead of $\neg(\neg\phi \wedge \neg\phi')$, $\exists r.\phi$ instead of $\geq_1 r.\phi$, and $\geq_n r$ instead of $\geq_n r.\phi$ if ϕ is $\top$.SHACL constraints are represented in the form of *(shape) constraints*, which are expressions of the form $s \leftarrow \phi$, with $s \in \mathbf{S}$ and ϕ a shape expression. A *shape atom* is an expression of the form $s(a)$, with s a shape name and a a node. A *shapes graph*[2] is a pair (C, T), where C is a set of shape constraints such that each shape name occurs exactly once on the left-hand side of a shape constraint, and T is a set of shape atoms, called *target set*, or simply *target*.

Non-recursive SHACL. We formally define non-recursive SHACL constraints as follows: a shape name s *directly refers* to a shape name s' in a set of constraints C, if C has a constraint $s \leftarrow \phi$ such that s' appears in ϕ. We say that s *refers* to s', if s directly refers to s', or there exists a shape name s'' such that s refers to s'', and s'' directly refers to s'. A set of SHACL constraints C is *non-recursive* if no shape name in C refers to itself.

Evaluation of shape expressions. A *(shape) assignment* for a data graph G extends G with a set L of shape atoms such that a occurs in G for each $s(a) \in L$. The evaluation of a shape expression ϕ over a data graph is defined in terms of a function $[\![\cdot]\!]^I$ that maps a (complex) shape expression to a set of nodes, and a property to a set of pairs of nodes. We refer to [3] for more details on the evaluation of shape expressions.

SHACL validation. There are two semantics for SHACL validation, the classical (or supported) model semantics from [6] and the stable model semantics from [3]. Here, we only present the supported model semantics. It was shown in [3] that both semantics coincide on non-recursive SHACL. Assume a SHACL document (C, T) and a data graph G. An assignment I for G is a *(supported) model* of C if $[\![\phi]\!]^I = s^I$ for all $s \leftarrow \phi \in C$. The data graph G *validates* (C, T) if there exists an assignment $I = G \cup L$ for G such that (i) I is a model of C, and (ii) $T \subseteq L$.

Normal Form. To ease presentation, in the rest of the paper we focus on *normalized* sets of SHACL constraint. That is, each SHACL constraint can have one of the following normal forms:

(NF1) $s \leftarrow \top$ *(NF2)* $s \leftarrow B$ *(NF3)* $s \leftarrow c$
(NF4) $s \leftarrow s_1 \wedge \cdots \wedge s_n$ *(NF5)* $s \leftarrow \neg s'$ *(NF6)* $s \leftarrow \geq_n r.s'$

It was shown in [3], that a set of constrains C can be transformed in polynomial time into a set of constraints C' in normal form such that for every data graph G and target T, G validates (C, T) if and only if G validates (C', T). Further, the normalization may introduce fresh shape names, but it can easily be shown that C' is non-recursive if C is non-recursive.

Example 1. Assume a shape StudentShape (left) and a data graph (right), written in Turtle syntax:

```
:StudentShape a sh:NodeShape ;            :Ben :enrolledIn :C1 .
    sh:targetNode :Ben ;
```

[2] `https://www.w3.org/TR/shacl/#shapes-graph`

```
    sh:property [
        sh:path :enrolledIn ;
        sh:qualifiedMinCount 1 ;
        sh:qualifiedValueShape [
        sh:class :Course ;]] .
```

The shape states that each $\mathsf{StudentShape}$ must be enrolled in at least one course and should be verified at node *Ben*. In the abstract syntax, we write the data graph $G = \{\mathit{enrolledIn}(\mathit{Ben}, C_1)\}$, the target $T = \{\mathsf{StudentShape}(\mathit{Ben})\}$, and C contains the constraint $\mathsf{StudentShape} \leftarrow \exists \mathit{enrolledIn}.\mathit{Course}$ The normalized version C' of C contains the constraints $\mathsf{StudentShape} \leftarrow \exists \mathit{enrolledIn}.s$ and $s \leftarrow \mathit{Course}$, where s is a fresh shape name. Clearly, extending G by assigning the shape name $\mathsf{StudentShape}$ to *Ben* does not satisfy the target $\mathsf{StudentShape}(\mathit{Ben})$, since *Ben* is not enrolled in any *Course*. Hence, G does not validate (C, T).

Answer Set Programming We introduce here some basic notation about Answer Set Programming (ASP) used throughout the paper and refer to [7] for more details on the language. We assume countably infinite, mutually disjoint sets $\mathbf{Preds} \supset \mathbf{C} \cup \mathbf{P}$ and $\mathbf{Var}$ of *predicate symbols*, and *variables*, respectively. A *term* is a variable from $\mathbf{Var}$ or a node from $\mathbf{N}$. The notion of an *atom* is extended from RDF atoms here to include expressions of the form $q(t_1, \ldots, t_n)$, where $q \in \mathbf{Preds}$ is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms; an atom is *ground* if its terms are nodes. A database is a set of ground atoms. An answer set program consists of a set of rules of the form $\psi \leftarrow \varphi$, where φ may be a conjunction of positive and negated atoms, and ψ is a (possibly empty) disjunction of atoms. We may call ψ the *head* of the rule and φ the *body* of the rule. We may write a rule $h_1, \ldots, h_n \leftarrow \varphi$ instead of a set of rules $h_1 \leftarrow \varphi, \ldots, h_n \leftarrow \varphi$. Roughly, a rule is satisfied by a database D in case the following holds: if there is a way to ground the rule by instantiating all its variables such that D contains the positive atoms in the body of the instantiated rule and does not contain the negative atoms, then it contains some atom occurring in the head of the rule. The semantics of answer set programs is given in terms of *stable models*. Intuitively, a stable model for (D, P), where D is a database and P a program, is a database D' that minimally extends D to satisfy all rules in P. We illustrate answer set programs with an example about 3-colorability.

Example 2. Let $\mathsf{D} = \{\mathsf{edge}(a, b), \mathsf{edge}(b, c), \mathsf{edge}(c, a),\ N(a), N(b), N(c)\}$ be a database storing a triangle over the nodes a, b, and c, and let P be a program with the following rules:

$$\begin{array}{ll} R(X) \vee B(X) \vee G(X) \leftarrow N(X) & \leftarrow \mathsf{edge}(X, Y), R(X), R(Y) \\ \leftarrow \mathsf{edge}(X, Y), B(X), B(Y) & \leftarrow \mathsf{edge}(X, Y), G(X), G(Y). \end{array}$$

P states that every node must be colored with red R, blue B, or green G and adjacent vertices must not be colored with the same color. Clearly, there are three possibilities to color the nodes and hence, three answer sets for (D, P) that minimally extend D to satisfy the rules. E.g. one stable model is $M = \mathsf{D} \cup \{R(a), B(b), G(c)\}$.

To implement the repair generator we selected Clingo[3], which provides additional features, like optimization functions, that will be present in the repair rules.

3 SHACL Repairs

In this section, we introduce the notion of repairs that we use in this work, analyze the kind of repairs that may be desirable in practice, and describe the design choices we will consider for the repair generator we propose. For repairs, we use the notion introduced in [2], where a repair is a set of facts that are added or removed from the input data graph so that the resulting graph validates the input shapes graph. We recall a slightly modified definition here.

Definition 1. *A* repair problem *is a tuple $\Psi = (G, C, T)$, where G is a data graph, and (C, T) is a shapes graph such that G does not validate (C, T). A* repair *for Ψ is a pair (A, D) of two sets of RDF atoms, where $D \subseteq G$, such that $(G \setminus D) \cup A$ validates (C, T).*

Note that the original definition of a repair problem in [2] includes a *hypothesis* set H, which allows to limit the space of possible additions by imposing the inclusion $A \subseteq H$. For simplicity, we do not limit the possible additions here, i.e. in the sense of [2], we simply let H to be the set of all possible RDF atoms.

When designing a repair generator, we need to make some choices. First, as also argued in [2], computing all possible repairs is not desirable: we naturally want the repairs to modify the data graph in a minimal way, i.e. additions and deletions that are not relevant for fixing the constraint violations should be excluded. For instance, the repair problem (G, C, T) in Example 1 can be solved, among other ways, by (i) adding to G the atom $\mathit{Course}(\mathit{C1})$, (ii) by adding to G the atoms $\mathit{Course}(\mathit{C2})$ and $\mathit{enrolledIn}(\mathit{Ben}, C2)$, or (iii) by adding to G the atoms $\mathit{Course}(\mathit{C1})$ and $\mathit{Course}(\mathit{C2})$. Observe that (i) is a repair that is minimal in terms of the *number* of modifications that are performed, i.e. cardinality-minimal. The repair (ii) can also be considered minimal, but in the sense of subset-minimality: observe that neither $\mathit{Course}(\mathit{C2})$ nor $\mathit{enrolledIn}(\mathit{Ben}, C2)$ *alone* suffice to fix the constraint violation. The repair (iii) is not minimal in either sense, because the addition of $\mathit{Course}(\mathit{C1})$ alone is sufficient to perform the repair.

Another issue is how to repair cardinality constraints of form (NF6). To satisfy them, we can either choose to generate new nodes, or we may try to reuse the existing nodes of the input data graph. There are scenarios where reusing nodes is not desired as we want to fix the violations while minimally changing the data graph. Reusing nodes may introduce wrong information from a real-world perspective and thus lower the quality of data. Consider the constraint $\mathsf{StudentShape} \leftarrow \exists \mathit{hasStudID}$ specifying that students must have a student ID and let the data graph have the atom $\mathit{hasStudID}(\mathit{Ben}, \mathit{ID1})$. To validate the target $\{\mathsf{StudentShape}(\mathit{Ann})\}$, a meaningful repair would be to generate a new

[3] `https://potassco.org/clingo/`

value $\_ID$ as placeholder and add $hasStudID(Ann, \_ID)$ instead of reusing $ID1$. Such placeholders can be replaced in a later step by the user with meaningful real-world values.

Unfortunately, forcing the repair generator to always introduce fresh values for cardinality constraints may sometimes leave out expected (minimal) repairs and even not produce any repairs at all. Consider the constraint $\mathsf{RegisteredCitizen} \leftarrow \exists MainAddress.Address \wedge \leq_1 MainAddress$, stating that registered citizens must have exactly one main address. Let $G = \{MainAddress(Ann, Ad1)\}$ and assume we want to validate that Ann is a $\mathsf{RegisteredCitizen}$. We may attempt to satisfy the constraint by adding the atoms $MainAddress(Ann, n)$ and $Address(n)$ for a fresh node n. However, then Ann would have two main addresses, which is not allowed. The only way to fix the violation is to reuse the node $Ad1$ and add $Address(Ann, Ad1)$ to the initial data graph. Also, for the repair problem from Example 1, mentioned above, by forcing to introduce fresh values we would miss the intuitive minimal repair that simply adds $Course(C1)$. In conclusion, there are scenarios where reusing existing nodes may be desired and even necessary. However, to preserve the quality of the data as much as possible, we want to prioritize the introduction of fresh values whenever possible and reuse existing constants only when necessary. We study both versions. More precisely, in Section 4, we propose a repair generator that always introduces fresh values and in Section 5 we present the extended version that allows to reuse constants, but introduces fresh values whenever possible.

4 Generating Repairs

In this section, based on existing works in databases by Bertossi et al. (see e.g. [4] and references therein), we present an encoding of the repair problem for non-recursive SHACL to ASP. We are especially interested in minimal repairs. To ease presentation, we describe here the encoding for a restricted setting, where only existential constraints of the form $s \leftarrow \geq_1 r.s'$, i.e., a special case of cardinality constraints of form (NF6), are allowed; we label them with (NF6$'$). In particular, rules will always introduce fresh values to repair existential constraints. We refer to Section 5 for the extension that support unrestricted cardinality constraints of form (NF6) and allows to reuse constants from the input.

4.1 Encoding into ASP

For a repair problem $\Psi = (G, C, T)$, where C is a set of non-recursive SHACL constraints in normal form, we construct a program P_Ψ, such that the stable models of (G, P_Ψ) will provide repairs for Ψ. Following the standard notation for repairs as logic programs in databases [4], to annotate atoms we will use special constants: (i) t^{**} intuitively states that the atom is true in the repair, (ii) t^* states that the atom is true in the input data graph or becomes true by some rule, (iii) t, states that the atom may need to be true and (iv) f states that the atom may need to be false. Intuitively, the repair program implements a top-down target-oriented approach, and starts by first making true all the shape atoms in the target. From this on, the rules for constraints specified by the shapes capture violations on the targets in the rule body and propose repairs in the rule head using the annotations described above. The rules will add annotated atoms which represent additions and deletions that can be applied to the data graph to fix the violations. Additions and deletions can interact, eventually stabilizing into a model that generates a (not necessarily minimal) repair.

For every constraint specified by a shape in the shapes graph, the repair program P_Ψ consists of four kinds of rules:

$P_{\text{Annotation}}$ consists of rules that collect existing atoms or atoms that are proposed to be in the repaired data graph.

P_{Repair} consists of rules that repair the constraints by proposing additions and deletions of atoms.

$P_{\text{Interpretation}}$ consists of rules that collect all the atoms that will be in the repaired data graph.

$P_{\text{Constraints}}$ consists of rules that filter out models that do not provide repairs.

We are ready to describe the repair program.

Adding the shape atoms in the target as facts. First, for each atom $s(a) \in T$, we add the rule $s\_(a, t^*) \leftarrow$, where $s\_$ is a fresh binary relation.

$P_{Annotation}$. For each class name B and property name p occurring in G and C, we create a new binary predicate $B\_$ and ternary predicate $p\_$, respectively. We add the following rules to P_Ψ:

$$\begin{array}{ll} B\_(X, t^*) \leftarrow B(X) & \qquad p\_(X, Y, t^*) \leftarrow p(X, Y) \\ B\_(X, t^*) \leftarrow B\_(X, t) & \qquad p\_(X, Y, t^*) \leftarrow p\_(X, Y, t) \end{array}$$

P_{Repair}. We present here the rules that participate in P_{Repair}. For each constraint $s \leftarrow \phi$ in C, we add specific rules that consider in the body the scenarios where s at a certain node is suggested to be true in the repair program or false, and propose in the head ways to make ϕ true or false, respectively. We note that the presence of negation in constraints may enforce that a shape atom is false at specific nodes. We present the repair rules for each normal form that ϕ can take, that is for each type of constraint of the form (NF1) to (NF6′) and add rules for both $s\_(X, t^*)$ and $s\_(x, f)$.

- If the constraint is of the form (NF1) or (NF3), then we do nothing here and treat them later as constraints.

- If ϕ is a class name B, that is of form (NF2), then we use the fresh binary predicate $B\_$ and add the rules:

$$B\_(X, t) \leftarrow s\_(X, t^*) \qquad\qquad B\_(X, f) \leftarrow s\_(X, f)$$

- If ϕ is of the form $s_1 \wedge \cdots \wedge s_n$, that is of form (NF4), then we use fresh binary predicates $s_{i}\_$ and add the rules:

$$s_1\_(X, t^*), \ldots, s_n\_(X, t^*) \leftarrow s\_(X, t^*) \qquad s_1\_(X, f) \vee \cdots \vee s_n\_(X, f) \leftarrow s\_(X, f)$$

- If ϕ is of the form $\neg s'$, that is of form (NF5), we add the rules:

$$s'\_(X,f) \leftarrow s\_(X,t^*) \qquad\qquad s'\_(X,t^*) \leftarrow s\_(X,f)$$

(*) If ϕ is of the form $\exists r.s'$, i.e., of form (NF6′), then we have to consider the scenarios where r is a property name p or an inverse property p^-. For the case where s is suggested to be true at X, i.e., for $s\_(X,t^*)$, we add a new p-edge from X to a fresh node and assign the node to s'. To this end, we use a function $@new(s,X,p)$, which maps a shape name s, a node X and a property name p to a new unique value Y. For the case where s is suggested to be false at X, i.e., for $s\_(X,f)$, we add disjunctive rules that, for all p-edges from X to some Y with s' true in Y, makes one of these atoms false. We add the rules for $r = p$. For $r = p^-$ the rules are analogously obtained by just swapping the variables in the argument of p.

$$s'\_(@new(s,X,p),t^*), p\_(X,@new(s,X,p),t) \leftarrow s\_(X,t^*)$$
$$p\_(X,Y,f) \vee s'\_(Y,f) \leftarrow s\_(X,f), p\_(X,Y,t^*)$$

$P_{Interpretation}$. For every class name B and property name p occurring in the input, we add the following rules:

$$B\_(X,t^{**}) \leftarrow B\_(X,t^*), not\ B(X,f)$$
$$p\_(X,Y,t^{**}) \leftarrow p\_(X,Y,t^*), not\ p(X,Y,f)$$

Intuitively, these rules will generate the atoms that will participate in the repaired data graph, that is the atoms that were added to the data graph, and those atoms from the data graph that were not deleted by the rules.

$P_{Constraints}$. We add to P_Ψ sets of rules that will act as constraints and filter out models that are not repairs.
(1) For each constraint of the form $s \leftarrow \top$, i.e., of form (NF1), we add $\leftarrow s\_(Y,f)$.
(2) For each constraint of the form $s \leftarrow c$, i.e., of form (NF3), we add the rules:

$$\leftarrow s\_(X,t^*), X \neq c \qquad\qquad \leftarrow s\_(c,f)$$

(3) For each class name B and property name p in the input, we add:

$$\leftarrow B\_(X,t), B\_(x,f) \qquad\qquad \leftarrow p\_(X,Y,t), p\_(X,Y,f)$$

Roughly, (1) and (2) ensure that models preserve constraints of type (NF1) and (NF3) which cannot be repaired, and (3) ensures that no atom is *both inserted and deleted* from G.

The atoms marked with t^{**} in a stable model of P_Ψ form a repaired data graph that validates (C,T).

Theorem 1. *Assume a repair problem $\Psi = (G,C,T)$. For every stable model M of (G,P_Ψ), the data graph G' validates (C,T), where G' is the set of all atoms of the form $B(a)$, $p(a,b)$ such that $B\_(a,t^{**})$ and $p\_(a,b,t^{**})$ are in M.*

We note that this theorem carries over to all the extensions and the version with cardinality constraints and constants we propose in this paper. It is easy to see that, since the rules are non-recursive in essence[4], the number of fresh nodes that can be introduced in a stable model is in the worst-case exponential in the size of the input constraints. However, we do not expect to see this behavior often in practice. We illustrate the repair program with a representative example.

Example 3. Consider the repair problem $\Psi = (G, C', T)$ from Example 1, where C' is the normalized version of C. We construct the repair program Π_Ψ as follows.

For $P_{\text{Annotation}}$ we use fresh predicates $\mathit{enrolledIn}\_$ and $\mathit{Course}\_$. The rules for $\mathit{Course}\_$ are $\mathit{Course}\_(X, t^*) \leftarrow \mathit{Course}\_(X, t)$, and $\mathit{Course}\_(X, t^*) \leftarrow \mathit{Course}(X)$; the rules for $\mathit{enrolledIn}\_$ are analogous. Intuitively, these rules will initially add the atom $\mathit{enrolledIn}\_(\mathit{Ben}, C_1, t^*)$ to the stable model. For P_{Repair}, we add the following rules, where F stands for the function $@\mathit{new}(\mathsf{StudentShape}, X, \mathit{enrolledIn})$.

$$\begin{aligned}
\mathit{enrolledIn}\_(X, F, t), s\_(F, t^*) &\leftarrow \mathsf{StudentShape}\_(X, t^*).\\
\mathit{enrolledIn}\_(X, Y, f) \vee s\_(Y, f) &\leftarrow \mathsf{StudentShape}\_(X, f), \mathit{enrolledIn}\_(X, Y, t^*)\\
\mathit{Course}\_(X, t) &\leftarrow s\_(X, t*)\\
\mathit{Course}\_(X, f) &\leftarrow s\_(X, f)
\end{aligned}$$

Intuitively, these rules together with the ones in $P_{\text{Annotation}}$ will add to the stable model the atoms $\mathit{enrolledIn}\_(\mathit{Ben}, \mathit{new}_1, t^*)$ and $\mathit{Course}\_(\mathit{new}_1, t^*)$, for a fresh node new_1. For $P_{\text{Interpretation}}$, we add: $\mathit{Course}\_(X, t^{**}) \leftarrow \mathit{Course}\_(X, t^*)$, $\mathit{not}\ \mathit{Course}\_(X, f)$ for $\mathit{Course}\_$ and proceed analogously for $\mathit{enrolledIn}\_$. For $P_{\mathit{Constraints}}$, we add the (constraint) rule: $\leftarrow \mathit{Course}\_(X, t), \mathit{Course}\_(X, Y, f)$ for $\mathit{Course}\_$ and proceed analogously for $\mathit{enrolledIn}\_$. Since no atom labelled with 'f' is generated by the rules, then the three atoms mentioned above will be annotated with 't^{**}' by the rules in $P_{\text{Interpretation}}$.

Thus, there is one stable model with the atoms $\mathit{enrolledIn}\_(\mathit{Ben}, C_1, t^{**})$, $\mathit{enrolledIn}\_(\mathit{Ben}, \mathit{new}_1, t^{**})$, and $\mathit{Course}\_(\mathit{new}_1, t^{**})$. The corresponding atoms $\mathit{enrolledIn}(\mathit{Ben}, C_1)$, $\mathit{enrolledIn}(\mathit{Ben}, \mathit{new}_1)$ and $\mathit{Course}(\mathit{new}_1)$ will form the repaired data graph G' that validates (C', T). Hence, the only repair is $(A, \emptyset)$, where A contains $\{\mathit{enrolledIn}(\mathit{Ben}, \mathit{new}_1)$ and $\mathit{Course}\_(\mathit{new}_1)\}$.

Additions and Deletions We want to represent repairs as sets of atoms that are added to and deleted from the input data graph. To achieve this, we use two fresh unary predicates add and del, and we add rules that "label" in a stable model with the label add all the atoms with 't^{**}' that were not in the data graph, and label with del all the atoms from the data graph that are annotated with 'f'. To this aim, for every class name B and property name p in the input, we introduce a function symbol (with the same name) whose arguments are the tuples of B and p, respectively. We show the rules for class names; the rules for property names are analogous.

[4] Technically speaking, the repair rules above may be recursive. However, if the annotation constants t, f, t^*, t^{**} are seen as part of the predicate's name (instead of being a fixed value in the last position), then the rules are non-recursive.

$$add(B(X)) \leftarrow B\_(X, t^{**}), not\ B(X) \qquad del(B(X)) \leftarrow B\_(X, f), B(X)$$

4.2 Generating Minimal Repairs

We are interested to generate cardinality-minimal repairs, i.e. repairs that make the least number of changes to the original data graph. More formally, given a repair $\xi = (A, D)$ for Ψ, ξ is *cardinality-minimal* if there is no repair $\xi' = (A', D')$ for Ψ such that $|A| + |D| > |A'| + |D'|$. As noted already in Section 3, repairs produced by our repair program built so far may not be cardinality-minimal, and this holds already for constraints without existential quantification. Consider the following example.

Example 4. Let (G, C, T) be a repair problem, where G is empty, $T = \{s(a)\}$, and C contains the constraints: $s \leftarrow s1 \vee s2$, $s1 \leftarrow A$, and $s2 \leftarrow A \wedge B$, where A and B are class names, and $s, s1, s2$ are shape names. To ease presentation, the constraints are not in normal form and $s1 \vee s2$ is a shortcut for $\neg(\neg s1 \wedge \neg s2)$. Clearly, to validate the target $s(a)$ the repair program will propose to make $s1(a)$ or $s2(a)$ true. Hence, there will be two stable models: one generates a repair that adds $A(a)$, i.e., contains $add(A(a))$, and the other adds both $A(a)$ and, the possibly redundant fact, $B(a)$, i.e., contains $add(A(a))$ and $add(B(a))$.

To compute cardinality-minimal repairs, which minimize the number of additions and deletions, we introduce a post-processing step for our repair program that selects the desired stable models based on a cost function. We count the distinct atoms for additions and deletions and add a cost off each of them. The repair program should only return stable models that minimize this cost. More specifically, we add the $\#minimize\{1, W : add(W); 1, V : del(V)\}$ optimization rule to P_Ψ, which uses a cost 1 for each addition or deletion. We can also change the cost for additions and deletions depending on different repair scenarios, where one could have a higher cost for additions over deletions or vice versa.

4.3 Repairing Maximal Subsets of the Target Set

In this section, we discuss the situation where it is not possible to repair all of the target shape atoms, e.g., because of conflicting constraints in shape assignments to these target shape atoms or because of unsatisfiable constraints. Consider for instance the constraint $s \leftarrow B \wedge \neg B$, where B is a class name. Clearly, there is no repair for any shape atom over s in the target, since there is no way to repair the body of the constraint. Similarly, consider the constraints $s1 \leftarrow B$ and $s2 \leftarrow \neg B$ and targets $s1(a)$ and $s2(a)$; in this case adding $B(a)$ violates the second constraint and not adding it violates the first constraint. In both scenarios, the repair program will return no stable model, and hence, no repair. However, it still might be possible to repair a subset of the shape targets. In practice, we want to repair as many targets as possible. To support such a scenario, we introduce the concept of *maximal repairs*, which is a relaxation of the previous notion of repairs.

Definition 2. *Let $\Psi = (G, C, T)$ be a repair problem. A pair (A, D) of sets of atoms is called a* maximal repair *for Ψ if there exists $T' \subseteq T$ such that (i) (A, D) is a repair for (G, C, T), and (ii) there is no $T'' \subseteq T$ with $|T''| > |T'|$ and (G, C, T'') having some repair.*

To represent this in the repair program, we add rules to non-deterministically select a target for repairing or skip a target if the repair program cannot repair it. This approach could be viewed similar in spirit to SHACL's *sh:deactivated* (`https://www.w3.org/TR/shacl/#deactivated`) directive that allows for deactivating certain shapes, with the difference that we "deactivate" *targets* instead of whole shapes which are automatically selected by the repair program based on optimization criteria.To this end, for each shape atom $s(a)$ in the input target set T, instead of adding all $s\_(a, t^*)$ as facts, we add rules to non-deterministically select or skip repair targets. If there are no conflicting or unsatisfiable constraints, then the stable models provide repairs for all the targets. However, if a repair of a target shape atom is not possible, because shape constraints advise t as well as f, then the repair program will skip this target shape atom and the stable models will provide repairs only for the remaining shape atoms in T. We introduce two predicates *actualTarget* and *skipTarget*, where *actualTarget* represents a shape atom in the target that will be selected to repair, whereas *skipTarget* represents a shape atom in the target that is skipped and will not be repaired. For each $s(a)$ in T we add the rules:

$$\mathit{actualTarget}(a, s) \vee \mathit{skipTarget}(a, s) \leftarrow s(a) \qquad s\_(a, t^*) \leftarrow \mathit{actualTarget}(a, s)$$

We want to first repair as many target shape atoms as possible, and then minimize the number of additions and deletions needed for these repairs. To this end, we add the $\#\mathit{minimize}\{1@3, X, s : \mathit{skipTarget}(X, s)\}$ optimization rule to P_Ψ to minimize the number of skipped targets and the $\#\mathit{minimize}\{1@2, W : \mathit{add}(W); 1@2, V : \mathit{del}(V)\}$ rule to minimize the additions and deletion. Note that we choose a higher priority level for minimizing the number of skipped targets (1@3) than for minimizing additions and deletions (1@2). This rule minimizes the *skipTarget* atoms and therefore maximizes the *actualTarget* atoms based on the cardinality.

Example 5. Let (G, C, T) be a repair problem, where $G = \{\mathit{enrolledIn}(\mathit{Ben}, C_1)\}$, $T = \{\mathsf{StudentShape}(\mathit{Ben}), \mathsf{TeacherShape}(\mathit{Ben})\}$, and C contains $\mathsf{TeacherShape} \leftarrow \exists \mathit{teaches} \wedge \neg\mathsf{StudentShape}$ and $\mathsf{StudentShape} \leftarrow \exists \mathit{enrolledIn}.\mathit{Course}$. Thus, Ben is a target node for both StudentShape and TeacherShape. However, the first constraint states that a node cannot be a TeacherShape and StudentShape at the same time, which causes a contradiction when applied to Ben. This causes the repair program to have no model. By applying the optimizations for maximal repairs, it will result in the target selection: $\mathit{actualTarget}(\mathit{Ben}, \mathsf{StudentShape})$, and $\mathit{skipTarget}(\mathit{Ben}, \mathsf{TeacherShape})$. The repair program skips the shape atom $\mathsf{TeacherShape}(\mathit{Ben})$, so that we at least have a repair for StudentShape. For this repair program, this is the maximum possible number of targets. Changing the optimization cost to skip targets allows to specify a preference among targets or shapes, thereby adapting to different repair scenarios.

5 Extension with Cardinality Constraints and Constants

In Section 4, we proposed a repair program for a restricted setting with cardinality constraints of the form $s \leftarrow \geq_n r.s'$ with $n = 1$. We now explain the extension to support cardinality constraints with unrestricted n, i.e., of form (NF6). In addition to supporting the generation of new values, we now also allow to reuse existing constants from the input, which may even be necessary to generate some repair. E.g., consider an empty data graph G, the set of constraints $C = \{s \leftarrow \exists p.s', s' \leftarrow c\}$, and the target $T = \{s(a)\}$. Since the second constraint forces the selection of the constant c when generating a value for p, the only possible repair for (G, C, T) is to add the atom $p(a, c)$. However, we prioritize picking a fresh node over an existing one if the latter is not necessary. We construct a repair program P'_Ψ for a repair problem $\Psi = (G, C, T)$ whose stable models provide repairs for Ψ. In particular, P'_Ψ contains all the rules from P_Ψ, except for the rules marked with (*) in P_{Repair}, i.e., the rules for existential constraints, which will be replaced by the rules described here.

Repairing cardinality constraints. If ϕ is of the form $\geq_n p.s'$, that is of form (NF6), then for repairing the case $s\_(X, t^*)$ we need to insert at least n p-edges to nodes verifying s'. We first collect all nodes from C that are part of constraints to make sure that all necessary nodes are available to be picked for additions of property atoms. For every node c in C, we add: $\mathit{const}(c) \leftarrow$

- For the case where s is suggested to be true at X, i.e., for $s\_(X, t^*)$, we add the following rules.

$$\mathit{choose}(s, X, p, 0) \vee \cdots \vee \mathit{choose}(s, X, p, n) \leftarrow s\_(X, t^*) \quad (2)$$

$$p\_(X, @\mathit{new}(s, X, p, 1..i), t) \leftarrow \mathit{choose}(s, X, p, i), i \neq 0 \quad (3)$$

$$0\ \{p\_(X, Y, t) : \mathit{const}(Y)\}\ |\mathit{const}(Y)| \leftarrow s(X, t^*) \quad (4)$$

$$n\ \{s'(Y, t^*) : p\_(X, Y, t^{**})\}\ \mathit{max}(n, |\mathit{const}(Y)|) \leftarrow s(X, t^*) \quad (5)$$

In the following, we explain the rules (2) - (5) in detail. For adding atoms over $p\_$ to satisfy $\geq_n p.s'$, we either generate fresh nodes using the function *@new* or we pick from collected constants. For generating atoms with fresh nodes, we add a disjunctive rule (2) and use a fresh *choose* predicate, which is used to non-deterministically pick a number from 0 up to n for adding atoms over $p\_$ to fix the cardinality constraint. To add the actual atoms, we add a rule (3) that produces this number of atoms using the *@new* function, which will generate a new unique value Y for every (s, X, p, i) tuple with s a shape name, p a property name, and $i \in \{1 \ldots n\}$. With these two rules, we can generate as many atoms over $p\_$ as necessary to satisfy the cardinality constraint. Similarly to adding atoms with fresh nodes, we can also pick constants from C. We add a rule (4) to pick a number of 0 up to the maximum number of constants – using Clingo's choice rules, which allow to be parameterised with a lower and upper bound of elements from the head to be chosen – which will only pick constants if either required, because of other constraints, or needed by the cardinal by adding optimization rules. In addition to adding atoms over p, we need to satisfy s' on a number of n nodes. We add a rule (5) to pick at least n, but might pick up to as many values as there are constants, so that we can satisfy the cardinality as well as any constraints that require specific constants. Note that an expression of the form l $\{W : V\}$ m intuitively allows to generate in the model a number between l and m W-atoms whenever V-atoms are also true.

- For the case where s is suggested to be false at X, i.e., for $s\_(X, f)$, we pick from all atoms $p(X, Y)$ to either delete the atom or falsify s' at Y. We add a disjunctive rule to pick one or the other (but not both).

$$\ell\ \{\psi_1 \vee \psi_2\}\ \ell \leftarrow s\_(X, f), \#count\{Y : p\_(X, Y, t^*)\} = m, m > (n-1)$$

where $\ell = m - (n-1)$, ψ_1 is the expression $p\_(X, Y, f) : p(X, Y), not\ s'\_(Y, f)$ and ψ_2 is $s'\_(Y, f) : p\_(X, Y, t^*), not\ p\_(X, Y, f)$. To make $s\_$ false at X, we have two disjunctive options that we can falsify. The first option is to falsify the $p\_$ atom. This can only be selected if $s'\_$ was not falsified at node Y. The second option is to falsify the $s'\_$ at node Y, which in return should only be possible if the $p\_$ atom was not falsified. By picking $m-(n-1)$ options, we make sure that only the maximum allowed cardinality will be in the repaired graph.

Constant Reuse Optimization. The rules above are allowed to pick from any constants that are needed to satisfy constraints in the current model. However, we want to pick a constant from C only if it is necessary to satisfy a constraint. To achieve this, for every constraint of the form $s \leftarrow \geq_n p.s'$, that is of form (NF6), we add the $\#minimize\{1@1, X, Y : p\_(X, Y, t), const(Y)\}$ optimization rule to P'_Ψ that minimizes the use of constants among the different minimal repairs. We choose a lower priority level (1@1) for minimizing the use of constants after minimizing additions and deletions with a priority (1@2) and after minimizing the number of skipped target atoms (1@3). We first want to have minimal repairs and then among them to pick the ones with minimal number of constants. Note that this encoding may produce different repairs from the encoding in Section 3 on the same example as illustrated below.

Consider again Example 3. The repair program P'_Ψ will generate three repairs. One repair will only add $Course(C_1)$. Intuitively, rule (2) adds $choose(s, X, p, 0)$ to the model, rule (3) and (4) will not add atoms, and rule (5) adds $s\_(C1, t^*)$ which together with the other rules treated in Example 3 add $Course\_(C1, t)$, and $Course(C1, t^{**})$, and hence . The second repair will add $enrolledIn(Ben, new_1)$ in addition to $Course(C_1)$ because of picking $i = 1$ in rule (2), and generating a fresh value new_1 in (3), and picking still $C1$ for $s\_$. The third repair will assign new_1 to $s\_$, thus resulting in the repair $(A, \emptyset)$ from Example 3. The optimization feature will return only the minimal repair that only adds $Course(C_1)$.

6 ASP Implementation

Repair Program. We developed a prototypical system for implementing SHACL repair programs using Java programming language and the ASP system Clingo.The prototype parses an RDF representation of a SHACL shapes graph and a data graph and transforms them into a repair program as a set of Clingo rules and facts. The repair program can then be executed using Clingo, which returns the stable models with (sub)sets of repaired shape target nodes and sets of additions and deletions as repairs for the data graph.

Unit Test Suite. To verify the implementation, we created a unit test suite with minimal examples that covers all the supported shape expressions. We grouped the test cases in four groups for class constraints, property constraints, value constraints and constraints with conflicts either within a shape or between multiple shape assignments. Each group includes expressions with conjunction, disjunction and negation. The unit test suite consist of a total of 43 test cases.

Data Shapes Test Suite. We applied the repair program to 16 selected test cases of the official SHACL data shapes test suite[5]. The selection was done based on the supported shape expressions of the repair program. All the selected tests were successful and repairs provided in the case of no conflicting constraints.[6]

7 Conclusion

We presented an approach to repair a data graph so that it conforms to a set of SHACL constraints. We first analyze the type of repairs that may be desirable in practice. To generate the repairs, inspired by existing work in databases, we encode the problem into an ASP program. We provide encodings for a restricted setting, which forces to introduce a new value to satisfy existential constraints, and for the extended setting that allows to reuse also existing constants. The optimizations as part of our approach were introduced with a view on practical scenarios, where we not only want to have minimal change, but also to avoid creating new data that is not sound from a real-world perspective. In case not all the shape targets can be repaired, we optimize to repair as many of them as possible. With the repair program and the ASP implementation, we have laid the foundation for bringing repairs into practical scenarios, and thus improving the quality of RDF graphs in practice.

Future work. Several tasks remain for future work. For the practical side, the next step will be to select use-cases where we can apply the repairs and evaluate the practical feasibility and explore repair quality and scalability. For the more technical direction, we plan to extend the approach to support SHACL *property paths*. Another direction is to support *recursive* SHACL constraints. They are also challenging because recursion combined with the introduction of fresh nodes, may cause non-termination of the repair process, i.e. an infinite repair might be forced. A related direction is to extend our approach to the so-called *class-based* and *property-based* targets. These targets bring implicit recursion to the repair problem, even when the constraints are non-recursive as in this paper, which makes dealing with such targets as challenging as the full recursive case.

Acknowledgements Supported by the Austrian Science Fund (FWF) and netidee SCIENCE project T 1349-N, and the FWF projects P30360 and P30873. Polleres is supported by funding in the European Commission's Horizon 2020 Research Program under Grant Agreement Number 957402 (TEAMING.AI).

[5] `https://w3c.github.io/data-shapes/data-shapes-test-suite/`

[6] Our prototype, test suites, and statistics are available from the authors upon request.

References

1. Shapes constraint language (SHACL). Tech. rep., W3C (Jul 2017), https://www.w3.org/TR/shacl/
2. Ahmetaj, S., David, R., Ortiz, M., Polleres, A., Shehu, B., Simkus, M.: Reasoning about explanations for non-validation in SHACL. In: Bienvenu, M., Lakemeyer, G., Erdem, E. (eds.) Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021. pp. 12–21 (2021). https://doi.org/10.24963/kr.2021/2, https://doi.org/10.24963/kr.2021/2
3. Andresel, M., Corman, J., Ortiz, M., Reutter, J.L., Savkovic, O., Šimkus, M.: Stable model semantics for recursive SHACL. In: Proc. of The Web Conference 2020. p. 1570–1580. WWW '20, ACM (2020). https://doi.org/10.1145/3366423.3380229, https://doi.org/10.1145/3366423.3380229
4. Bertossi, L.: Database Repairing and Consistent Query Answering. Morgan & Claypool Publishers (2011). https://doi.org/https://doi.org/10.2200/S00379ED1V01Y201108DTM020
5. Bertossi, L.E.: Database Repairing and Consistent Query Answering. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2011). https://doi.org/10.2200/S00379ED1V01Y201108DTM020, https://doi.org/10.2200/S00379ED1V01Y201108DTM020
6. Corman, J., Reutter, J.L., Savkovic, O.: Semantics and validation of recursive SHACL. In: Proc. of ISWC'18. Springer (2018). https://doi.org/10.1007/978-3-030-00671-6_19, https://doi.org/10.1007/978-3-030-00671-6_19
7. Eiter, T., Ianni, G., Krennwallner, T.: Answer set programming: A primer. In: Tessaris, S., Franconi, E., Eiter, T., Gutiérrez, C., Handschuh, S., Rousset, M., Schmidt, R.A. (eds.) Reasoning Web. Semantic Technologies for Information Systems. Lecture Notes in Computer Science, vol. 5689, pp. 40–110. Springer (2009). https://doi.org/10.1007/978-3-642-03754-2_2, https://doi.org/10.1007/978-3-642-03754-2_2
8. Gayo, J.E.L., Prud'hommeaux, E., Boneva, I., Kontokostas, D.: Validating RDF Data. Synthesis Lectures on the Semantic Web: Theory and Technology, Morgan & Claypool Publishers (2017). https://doi.org/10.2200/S00786ED1V01Y201707WBE016, https://doi.org/10.2200/S00786ED1V01Y201707WBE016
9. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot asp solving with clingo. Theory and Practice of Logic Programming **19**(1), 27–82 (2019). https://doi.org/10.1017/S1471068418000054
10. Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutiérrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., Ngomo, A.N., Polleres, A., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., Zimmermann, A.: Knowledge Graphs. Synthesis Lectures on Data, Semantics, and Knowledge, Morgan & Claypool Publishers (2021). https://doi.org/10.2200/S01125ED1V01Y202109DSK022, https://doi.org/10.2200/S01125ED1V01Y202109DSK022
11. Marileo, M.C., Bertossi, L.E.: The consistency extractor system: Answer set programs for consistent query answering in databases. Data Knowl. Eng. **69**, 545–572 (2010). https://doi.org/http://dx.doi.org/10.1016/j.datak.2010.01.005