A Logic Programming Approach to Repairing SHACL Constraint Violations

Shqiponja Ahmetaj
TU Wien, Austria
Vienna University of Economics and Business, Austria

Robert David[1]
Vienna University of Economics and Business, Austria
Semantic Web Company GmbH, Vienna, Austria

Axel Polleres
Vienna University of Economics and Business, Austria
Complexity Science Hub, Vienna, Austria

Mantas Šimkus
TU Wien, Austria
Umeå University, Sweden

Abstract

The Shapes Constraint Language (SHACL) is a recent standard, a W3C recommendation, for *validating* RDF graphs against *shape* constraints to be checked on *target nodes* of a data graph. The standard also describes the notion of *validation reports*, which detail the results of the validation process. In case of violation of constraints, the validation report should explain the reasons for non-validation, offering guidance on how to identify or fix violations in the data graph. 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, where the answer sets contain (minimal) repairs. We then study a scenario where it is not possible to simultaneously repair all the targets, which may be 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. Finally, we add support for repairing constraints which use property paths and equality of paths. Our implementation in clingo is – to the best of our knowledge – the first implementation of a repair program for SHACL.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Logic programming and answer set programming

Keywords and phrases SHACL, Shapes Constraint Language, Database Repairs, Knowledge Graphs, Semantic Web, Answer Set Programming

Digital Object Identifier 10.4230/TGDK.3.3.1

Related Version *Previous Version*: `https://iswc2022.semanticweb.org/wp-content/uploads/2022/11/978-3-031-19433-7_22.pdf`

Supplementary Material *Software (Source Code)*: `https://github.com/robert-david/shacl-repairs` [23] archived at `swh:1:dir:9de47c8b97cba079add751f9e2a64421238a67f3`

Funding This work was supported by the Austrian Science Fund (FWF) Cluster of Excellence "BILAI" [10.55776/COE12]. Ahmetaj was also supported by the Austrian Science Fund (FWF) projects netidee SCIENCE [T1349-N] and the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201].

Received 2024-07-19 **Accepted** 2025-09-12 **Published** 2025-12-10

[1] Corresponding author

1 Introduction

Semantic Web standards provide means to represent and link heterogeneous data sources in knowledge graphs [30], thereby potentially solving common data integration problems. Indeed, this approach became increasingly popular in enterprises as *enterprise knowledge graphs (EKG)*, which integrate data silos of an enterprise into one consolidated knowledge graph [27]. 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 consolidated knowledge graphs. Although our work is motivated by the high quality requirements of EKGs in practice, it is by far not limited to this scenario and generally applicable to graph data. 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 [32]. In this setting, the validation requirements are specified in a *shapes graph* (C, T) that consists of a collection C of constraints and a specification T of nodes, called *targets*, to which the constraints should be applied.

The result of validating an RDF graph G against a shapes graph (C, T) is presented as a *validation report*, which indicates whether the input graph has passed the validation test. In the event of constraint violations, the report should provide details explaining the issues and violations found during the validation process. However, the SHACL standard offers limited guidance regarding the specific content of validation reports, except for the requirement that these reports must include information about the violated nodes, constraints, and the affected part of the input graph. We follow the proposal of [3] to define *explanations* for SHACL non-validation in the style of database *repairs* [7], namely as a set of additions and deletions of facts to the input data graph that will cause a previously invalid setting to become valid. Consider for instance a requirement, described as a shape constraint StudentShape, stating that persons need to be enrolled in at least one course to qualify as students and the target requires to validate StudentShape for Ben. In the abstract syntax, we define C to contain the constraint $\mathsf{StudentShape} \leftarrow \exists \mathit{enrolledIn}.\mathit{Course}.$ and T to be $\{\mathsf{StudentShape}(\mathit{Ben})\}$. In case the shapes graph (C, T) is applied to the data graph $G = \{\mathit{enrolledIn}(\mathit{Ben}, C_1)\}$, stating that Ben has the property $enrolledIn$, a violation will be reported. An intuitive explanation for the violation is that there is no $Course$-fact about node C_1, and a repair would add the fact $\mathit{Course}(C_1)$ to the data graph G. Indeed, in many common scenarios for knowledge graphs (like the automated integration of heterogeneous data sources) inconsistencies might appear frequently and can be mitigated using SHACL validation [28]. Consequently, there is a need to *automatically* identify repairs that can be applied to the data graph in order to achieve consistency.

The main contributions of our work are as follows:

- We propose to compute repairs of a data graph by encoding the problem into *Answer Set Programming (ASP)* [25], a formalism sufficiently expressive to capture the high complexity of the repair problem for SHACL [3]. 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 (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 decided for clingo [29]), 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 to the data graph during the repair process, an infinite number of repairs is possible. This needs to be handled carefully, and several design choices have to be made.
- For presentation clarity, we initially present a *basic encoding* of the repair task into ASP, making the core idea of our approach easily accessible. In this encoding, the repair program tries to find a repair of the input shapes graph for a basic fragment of SHACL that excludes

qualified number restrictions and complex paths but allows existential quantification. It also supports target declarations for nodes, classes, and the domain and range of properties. 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 node is always introduced. We argue that this is a reasonable approach in general and closely aligns with the standard notion of *Skolemization*. Furthermore, leveraging the optimization features of ASP, our repair program ensures that among all possible repairs it can produce, only those that are minimal in terms of cardinality are returned. This guarantees that constraint violations are resolved with the fewest possible changes to the data graph.

- There are scenarios where enforcing the repair program to always introduce fresh values for satisfying cardinality constraints may exclude certain expected minimal repairs or even prevent any repairs from being generated. In some cases, reusing existing nodes is not only desirable, but also necessary. However, to maintain data quality as much as possible, our approach prioritizes introducing fresh values whenever feasible, resorting to existing constants only when necessary. We then present an extended version of our basic encoding that supports reusing existing nodes while still favoring the introduction of fresh values whenever possible. Moreover, we extend the fragment from the basic encoding to incorporate other SHACL features such as *cardinality constraints*, *constants*, and *property paths*. In particular, we add support for sequential and inverse paths, as well as equality constraints over paths and properties.
- We observe that requiring a repair to resolve violations for *all* targets specified in the input may be too strong. If the data graph has one inherently unsatisfiable target (e.g., because of some conflicting or contradictory constraints), then the repair program will not have any models 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 aims to repair as many targets as possible. We show how our encoding can be extended to generate repairs under this new notion by leveraging the optimization features of clingo and incorporating disjunctive rules that allow certain targets to be skipped when necessary.
- We have implemented and tested these encodings using the clingo ASP system, which showed that our approach is promising for managing and improving the quality of RDF graphs in practice. More precisely, we used the Java programming language to implement the translation of the SHACL repair programs, which can be executed using clingo. We tested the implementation using a set of unit tests and a subset of the data shapes test suite to test the functional correctness and performance tests for Wikidata as a real-world data set.

The repair program and the ASP implementation presented in this paper establish a foundation for repairing RDF graphs in the presence of SHACL constraints, laying the groundwork for practical tools that enhance data quality in real-world applications. A preliminary version of this work has been published in [4]. This journal version builds upon [4] with the following key contributions: (i) an expanded SHACL fragment that now supports (limited) path expressions and equality over paths, covering a substantial portion of SHACL Core constraint components; (ii) broader support for SHACL targets, extending beyond node targets to include class-based and property-based targets, subject to minor syntactic restrictions that ensure the fragment remains non-recursive and avoids cyclic dependencies between constraints and targets; (iii) proofs for the proposed rewritings; and (iv) a real-world evaluation using Wikidata.

1.1 Related Work

There is a large body of work in the area of databases on repairing integrity constraints violations; for an overview of this topic see [14]. The closest in spirit to our work is [35], which specifies database repairs in the presence of integrity constraints using disjunctive logic programs with the

answer set semantics. These repairs minimally modify a database to achieve conformance with a set of integrity constraints. More specifically, given an inconsistent database instance D and a set of integrity constraints, [35] proposes a repair program, whose stable models are in a one-to-one correspondence with the repairs of D. We follow this general idea for the graph-based RDF data model scenario in the presence of SHACL constraints. The adaptation is challenging because SHACL includes features not present in traditional integrity constraints, such as unrestricted negation in the rule body, which carries significant semantic consequences.

In the context of Description Logic, there is extensive research on repairing the data (called ABox) so that certain unwanted consequences from a knowledge base are removed. Baader et al. study optimal repairs [10, 11, 12], which repair the ABox while preserving as much as possible from the consequences of an ontology (or TBox). The TBox is assumed to be static and cannot be changed in the repair process. They also look at optimal repairs in the context of contractions in belief change [8, 9, 13]. We follow a similar approach in the sense of assuming SHACL constraints to be carefully designed and without errors, and focusing only on repairing the data graph.

Also related to our work is [20], where Calvanese et al. focus on explanations for non-entailment of a given tuple from a knowledge base over Description Logic ontologies. Explanations in this setting are considered as minimal sets of facts such that, if added to the input data, the query would be entailed, while the knowledge base would remain consistent with the added information. Similar to our work, the explanation problem is formalized as an abductive task and considers preferences, where subset-minimal and cardinality-minimal explanations and their complexity are studied. Analogously, [21] addresses the problem of explaining why a query is not entailed under existential rules.

A large body of works on so-called *consistent query answering* has developed since the seminal paper [7]. The idea is to exploit the repairs of an inconsistent database to provide answers to a query. Various inconsistency-tolerant semantics based on what repairs to select for accepting query answers have been studied in various database and knowledge representation settings [7, 14, 33, 17, 35], including implemented systems [35, 16, 15]. A recent work [5] investigates consistent query answering under SHACL constraints, using a core fragment of SPARQL – the standardized language for querying RDF data – as the query language. Although our work does not yet address queries, the repair implementation we provide represents a crucial first step toward enabling consistent query answering in this context.

Further related but orthogonal work to ours involves axiom pinpointing, which focuses on identifying minimal sets of axioms sufficient to entail (or not entail) a specific expression. Such sets are called justifications and represent explanations of an entailment. An overview of axiom pinpointing is provided by Peñaloza [37] and Kalyanpur et al. [31] presents work on computing justifications in the context of Description Logic ontologies. There are works that study provenance for database query results, which originated in database theory and was extended to the Description Logic setting in the context of Ontology-based Data Access (OBDA) [36, 18]. Li et al. [34] propose to repair Description Logic ontologies using a combination of axiom weakening and completing. This approach aims to strike a balance between minimizing the negative impacts of removing unwanted axioms while preserving correct consequences. Work on static verification [19, 2] study the problem of verifying if a sequence of updates (additions and deletions) applied to an input data graph will preserve or violate a set of constraints expressed as Description Logic ontologies.

1.2 Organization of the Paper

The paper is structured as follows.

- Section 2 describes SHACL validation, normalization of SHACL constraints and answer set programming, which represent the foundations of our repair approach and the implementation.

- Section 3 introduces our notion of repairs for SHACL non-validation and the design choices we consider for our repair program.
- Section 4 provides an encoding of the repair program into an ASP program. We provide rules for generating repairs and optimizing them to be of a small size. In additions, in case not all targets can be repairs, we suggest a more relaxed version that repairs maximal subsets of targets.
- Section 5 extends this encoding to repair constraints with arbitrary cardinality and to allow the reuse of existing constants in this case.
- Section 6 further extends this cardinality constraint encoding by supporting the repair of SHACL property paths and equality of paths.
- Section 7 describes the ASP implementation and includes the translation of SHACL shapes from RDF Turtle syntax into our abstract syntax.
- Section 8 tests this implementation. We present our test scenarios, which are unit tests, test cases based on the official SHACL data shapes test suite and performance tests based on a real-world scenario over Wikidata, and report the results.
- Section 9 concludes the paper by summarizing the contributions and proposing directions for future work.

2 SHACL Validation and Answer Set Programming

In this section, we describe SHACL and the notion of *validation* against RDF graphs. The Shapes Constraint Language SHACL [32] is the W3C recommendation for validating graphs against sets of constraints. A SHACL processor can validate a *data graph* against constraints in a *shapes graph* and in the case of violations provide details to the user as part of a validation report. For an introduction to data validation, SHACL, and its close relative ShEx, we refer to [28, 1]. To ease presentation, SHACL constraints will be represented using an abstract syntax and normal form, which will also serve as a basis for implementation. We also describe answer set programming (ASP), which we use to implement the repair program and the notation for ASP that we use in the paper.

2.1 SHACL Validation

We use the abstract syntax from [3] for RDF and SHACL. Note that in this work we focus on the "Core Constraint Components" of SHACL with restricted path expressions.

Data graphs

We first define *data graphs*[2], which are RDF graphs to be validated against shape constraints. Assume countably infinite, mutually disjoint sets N_N, N_C, and N_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 N_S be a countably infinite set of *shape names*, disjoint from N_N, N_C, and N_P. A *path expression* E is a regular expression built from the operator $\cdot$, symbols in N_P, and expressions p^- for each $p \in N_P$, where p^- is the inverse property of p.

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

A *shape expression* ϕ is of the form:

$$\phi, \phi' ::= \top \mid s \mid B \mid c \mid \phi \wedge \phi' \mid \neg\phi \mid \geq_n E.\phi \mid E = p$$

where $s \in N_S$, $B \in N_C$, $c \in N_N$, n is a positive integer, E is a path expression and $p \in N_P$. In what follows, we may write $\phi \vee \phi'$ instead of $\neg(\neg\phi \wedge \neg\phi')$, $\exists E.\phi$ instead of $\geq_1 E.\phi$, and $\geq_n E$ instead of $\geq_n E.\phi$ if ϕ is $\top$. SHACL constraints are represented in the form of *(shape) constraints*, which are expressions of the form $s \leftarrow \phi$, with $s \in N_S$ 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 *target* is an expression 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*,
- expression of the form $\exists p$ with $p \in N_P$, also called *subjects-of target*,
- expression of the form $\exists p^-$ with $p \in N_P$, also called *objects-of target*.

A *shapes graph*[3] 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 targets. Informally, we call a set of constraints C *recursive* if there is a shape name s that directly or indirectly refers to itself in the shapes constraints. In the present paper, we assume non-recursive constraints.

Evaluation of shape expressions

A *(shapes) assignment* for a data graph G is an expansion $I = G \cup L$ of G with a set L of shape atoms such that a occurs in G for each $s(a) \in L$. We denote with $V(G)$ (resp. $V(I)$) the set of nodes appearing in G (resp. I). The evaluation of a shape expression ϕ over an assignment I is defined in terms of an 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 a set of nodes $[\![\phi]\!]^I \subseteq V(I)$.

$$[\![\top]\!]^I = V(G) \qquad [\![s]\!]^I = \{a \mid s(a) \in I\}$$
$$[\![B]\!]^I = \{a \mid B(a) \in I\} \qquad [\![c]\!]^I = c$$
$$[\![\phi_1 \wedge \phi_2]\!]^I = [\![\phi_1]\!]^I \cap [\![\phi_2]\!]^I \qquad [\![\neg\phi]\!]^I = V(I) \setminus [\![\phi]\!]^I$$
$$[\![p]\!]^I = \{(a,b) \mid p(a,b) \in I\} \qquad [\![p^-]\!]^I = \{(a,b) \mid p(b,a) \in I\}$$
$$[\![\geq_n E.\phi]\!]^I = \{a \mid \{(a,b) \in [\![E]\!]^I \text{ and } b \in [\![\phi]\!]^I\} \geq n\} \qquad [\![E \cdot E']\!]^I = [\![E]\!]^I \circ [\![E']\!]^I$$
$$[\![E = p]\!]^I = \{a \mid \forall b : (a,b) \in [\![p]\!]^I \text{ iff } (a,b) \in [\![E]\!]^I\}$$

SHACL validation

We now present the semantics for validation of a shapes graph (C, T) by a data graph G. There are several validation semantics for SHACL [22, 6], which coincide on non-recursive SHACL. Here, we only present the supported model semantics [22]. In particular, we require that all the nodes that occur in C and T appear also in G.

▶ **Definition 1.** *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 for G such that (i) I is a model of C, and (ii) $[\![W]\!]^I \in s^I$ for every target $(W, s) \in T$.*

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

Normal Form

To ease presentation, in the rest of the paper we focus on *normalized* sets of SHACL constraints. That is, each SHACL constraint is constructed using the following normal forms:

$$\textit{(NF1)}\ s \leftarrow \top \qquad \textit{(NF2)}\ s \leftarrow B \qquad \textit{(NF3)}\ s \leftarrow c$$
$$\textit{(NF4)}\ s \leftarrow s_1 \wedge \cdots \wedge s_n \qquad \textit{(NF5)}\ s \leftarrow \neg s' \qquad \textit{(NF6)}\ s \leftarrow \geq_n E.s' \qquad \textit{(NF7)}\ s \leftarrow E = p$$

It was shown in [6] that a set of constraints 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). Note that the normalization process recursively introduces fresh shape names for sub-expressions until the normal form is reached. Moreover, if the input shapes graph is non-recursive, then clearly the normalized one is also non-recursive.

▶ **Example 2.** Assume a shapes graph with shape constraints for StudentShape (left) and a data graph (right), written in Turtle syntax.

```
:StudentShape a sh:NodeShape;
    sh:targetNode :Ben;
    sh:property [
        sh:path :enrolledIn;
        sh:qualifiedMinCount 1;
        sh:qualifiedValueShape [
            sh:class :Course;
        ]
    ] .
```

```
:Ben :enrolledIn :C1 .
```

The shapes graph states that each node conforming to StudentShape must be enrolled in at least one course and it should be verified at node *Ben*. The target is represented in the Turtle syntax by the *sh:targetNode* property, and the statements below *sh:property* define the constraints to be verified on the node *Ben*. The constraints specify via *sh:path* to check whether there is an *enrolledIn* outgoing edge from target Ben to at least 1 node (*sh:qualifiedMinCount*) that is an instance of the *Course* class, specified with *sh:class*. In the abstract syntax, we write the data graph G, set of constraints C, and targets T as follows:

$$G = \{\mathit{enrolledIn}(\mathit{Ben}, C_1)\},$$
$$C = \{\mathsf{StudentShape} \leftarrow \exists \mathit{enrolledIn}.\mathit{Course}\},$$
$$T = \{\mathsf{StudentShape}(\mathit{Ben})\}$$

The normalized version C' of C contains the constraints

$$\mathsf{StudentShape} \leftarrow \exists \mathit{enrolledIn}.s,$$
$$s \leftarrow \mathit{Course}$$

where s is a fresh shape name. Clearly, extending G by assigning the shape name StudentShape to *Ben* does not satisfy the target StudentShape(*Ben*), since *Ben* is not enrolled in any (node of class) *Course*. Hence, G does not validate (C, T).

We note that the translation of SHACL shapes represented in RDF syntax into the abstract syntax is described in Section 7 as part of the ASP implementation.

2.2 Answer Set Programming

Answer Set Programming (ASP) is a declarative problem solving paradigm based on Logic Programming and Nonmonotonic Reasoning [25]. The idea of ASP is to describe (or model) a problem as a nonmonotonic logic program and let an ASP solver compute the solutions to the problem. The program consists of logic rules (with head and body), facts (head without a body) and constraints (body without a head). ASP has extensions for negation and disjunction in rule heads for modeling nonmonotonic programs. The ASP paradigm lets users focus on describing the problem itself instead of coding a solution for it, which is computed by an ASP solver. The solutions, so-called answer sets or stable models, are minimal models of a logic program. There can be none, one, or multiple stable models for a program.

The implementation of the repair program is done by generating an ASP program from the SHACL shapes and the data graph. By building on the minimality of ASP models, the repair program provides minimal repairs as part of the minimal models.

We introduce here some basic notation about Answer Set Programming (ASP) used throughout the paper and refer to [25] for more details on the language. We assume countably infinite, mutually disjoint sets $\mathsf{Preds} \supset N_C \cup N_P$ and Var of *predicate symbols*, and *variables*, respectively. A *term* is a variable from Var or a node from N_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 $q \in \mathsf{Preds}$ 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 (G, P), where G is a data graph and P a program, is a database D that minimally extends G to satisfy all rules in P. We illustrate answer set programs with an example about 3-colorability.

▶ **Example 3.** Let $G = \{\mathsf{edge}(a, b), \mathsf{edge}(b, c), \mathsf{edge}(c, a), N(a), N(b), N(c)\}$ be a data graph storing a triangle over the nodes a, b, and c, and let P be a program with the following rules:

$$R(X) \vee B(X) \vee G(X) \leftarrow N(X) \qquad \leftarrow \mathsf{edge}(X, Y), R(X), R(Y)$$
$$\leftarrow \mathsf{edge}(X, Y), B(X), B(Y) \qquad \leftarrow \mathsf{edge}(X, Y), G(X), G(Y).$$

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 (G, P) that minimally extend G to satisfy the rules. E.g., one stable model is $M = G \cup \{R(a), B(b), G(c)\}$.

To implement the repair program we selected the clingo ASP system for grounding and solving logic programs [29], which is available online.[4] Clingo provides additional features, like optimization functions, that will be present in the repair program and they will be explained when used by the specific rules.

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

3 Exploring Design Choices for 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 program we propose. For formally defining repairs, we use the notion of explanations for non-validation in SHACL introduced in [3], 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 4.** *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 [3] includes a *hypothesis* set H given in the input, which allows to limit the space of possible additions by imposing the inclusion $A \subseteq H$. In this work, we do not consider a given input set H of hypotheses and thereby do not limit the possible additions in such a way. Instead, we assume that H consists of all the atoms that can be formed from the class and property names, the nodes occurring in the input, and a finite set of fresh nodes that will be introduced by the repair rules. Since the input constraints are non-recursive, it can be easily argued that the number of fresh nodes introduced by the repair program is (possibly exponentially) bound by the number of constraints. Thus, in what follows, we omit H from the input and write (G, C, T) for a repair problem. In particular, when a property atom needs to be added in the repair, we prefer to introduce fresh nodes instead of reusing nodes from the input.

When designing a repair program, we need to make some choices. First, as also argued in [3], 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 2 can be solved, among other ways, by (i) adding to G the atom $Course(C1)$, (ii) by adding to G the atoms $Course(C2)$ and $enrolledIn(Ben, C2)$, or (iii) by adding to G the atoms $Course(C1)$ and $Course(C2)$. Observe that (i) is a repair that is minimal in terms of the *number* of changes 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 $Course(C2)$ nor $enrolledIn(Ben, C2)$ *alone* suffice to fix the constraint violation. The repair (iii) is not minimal in either sense, because the addition of $Course(C1)$ alone is sufficient to perform the repair.

Another issue is how to repair cardinality constraints of the form (NF6). To satisfy them, we can either choose to generate fresh 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

$$\mathsf{StudentShape} \leftarrow \exists hasStudentID$$

specifying that students must have a student ID and let the data graph include the atom $hasStudentID(Ben, ID1)$. To validate the target $\mathsf{StudentShape}(Ann)$, a meaningful repair would be to generate a new value $\_ID$ as placeholder value and add $hasStudentID(Ann, \_ID)$ instead of reusing $ID1$. Reusing $ID1$ and adding $hasStudentID(Ann, ID1)$ would assign Ben's student ID to Ann, which is likely to be a problem from a real-world perspective (or, resp., might violate further constraints in a more elaborate scenario). Indeed, constructing an entirely new value that does not yet exist in the data graph comes without any underlying semantics beyond what the shape constraints describe, and therefore avoids unintentional (re)use from a real-world perspective. Such placeholders can be replaced in a later step by the user with meaningful real-world values.

Unfortunately, in turn forcing the repair program to always introduce fresh values for cardinality constraints may sometimes leave out expected (minimal) repairs and may even not produce any repairs at all. Consider the constraint

$$\mathsf{StudentShape} \leftarrow \exists \mathit{hasStudentID} \wedge \leq_1 \mathit{hasStudentID}$$

stating that students must have exactly one student ID. Let $G = \{\mathit{hasStudentID}(\mathit{Ann}, \mathit{ID}1)\}$ and assume we want to validate that Ann conforms to StudentShape. We may attempt to satisfy the constraint by adding the atoms $\mathit{hasStudentID}(\mathit{Ann}, \mathit{new}_1)$ for a fresh node new_1. However, then Ann would have two main addresses, which is not allowed. The only way to fix the violation is to reuse the node $\mathit{ID}1$.

Also, for the repair problem from Example 2, mentioned above, by forcing to introduce fresh values we would miss the intuitive minimal repair that simply adds $\mathit{Course}(C1)$. 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 program 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.

Also, Section 6 further extends the repair program to support repairs for path expressions E, which requires to add property atoms to generate property paths for additions and to delete property atoms to cut property paths for deletion.

4 Generating Repairs

In this section, we present an encoding of the repair problem to ASP that captures the violations and proposes changes to the data graph so that the violations are fixed. It is given in terms of an answer set program whose stable models will provide repairs as a set of additions and deletions to the input data graph. We are especially interested in cardinality-minimal repairs, that can be generated by the program. We do not consider subset-minimal repairs here and leave it for future work.

To ease presentation, we describe here the encoding for a restricted setting, where only existential constraints of the form $s \leftarrow \geq_1 p.s'$, 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 and allows the reuse of constants from the input. Note that we also do not consider here constraints of the form (NF7). The unrestricted version of (NF6) and constraints of the form (NF7) will be treated in Section 6.

For the ASP encoding, we focus on shapes graphs (C, T), where C contains non-recursive constraints and T only contains node targets of the form $s(c)$. As for the latter, we can convert the other types of targets into node targets by simply *grounding* them w.r.t. the input data graph. More specifically, for a data graph G and an arbitrary shapes graph (C, T), we can generate an equivalent shapes graph (C, T_{gr}) over only node targets, where

$$T_{gr} = \{(c, s) \mid c \in [\![W]\!]^G \text{ for each target } (W, s) \in T\}$$

is the grounding of T w.r.t. G. It is easy to verify, that G validates (C, T) if and only if G validates (C, T_{gr}). However, this equivalence may not hold when considering repairs which change the input data graph and may update the grounding of the targets. In particular, the grounding may need to be recomputed after a repair is applied, as illustrated in the following example.

▶ **Example 5.** Consider a data graph $G = \{B_1(a), B_1(b), B_2(b)\}$ and the shapes graph (C, T), where C has the constraint $s \leftarrow B_2$ and the target is $T = \{(B_1, s)\}$. Clearly, G does not validate (C, T), and hence neither the corresponding shapes graph (C, T_{gr}), where $T_{gr} = \{(a, s), (b, s)\}$ is the grounding of T w.r.t. G. Consider the repair $R = (\emptyset, \{B_1(a)\})$, which removes only the fact $B_1(a)$ from G. Clearly, R is a repair for (G, C, T) since $G' = G \setminus \{B_1(a)\}$ validates (C, T), but it is not a repair for (G, C, T_{gr}). Clearly, every repair for (G, C, T_{gr}) is also a repair for (G, C, T).

Intuitively, the above issues appear because the grounding of the targets may change depending on the repair. If the repair modifies how class or property names in the targets are grounded, then a repair that works for the grounded version of the shapes graph may not necessarily be valid for the ungrounded version and vice versa.

▶ **Proposition 6.** *Let (C, T) be a shapes graph. Then, for every data graph G and pair $R = (A, D)$ that does not use the class and property names appearing in T, it holds that R is a repair for (G, C, T_{gr}) iff R is a repair for (G, C, T), where T_{gr} is the grounding of T w.r.t. G.*

To ensure that the repairs obtained from the ASP encoding do not use the class and property names appearing in the targets, it suffices to restrict the input shapes graphs (C, T) to not allow the shapes names in the constraints C to use the class and property names appearing in the targets T. We say that a shape name s *directly uses* ℓ, where ℓ is a shape, class, or property name, if there exists a constraint $s \leftarrow \phi$ in C such that ℓ occurs in ϕ. We say that a shape name s *uses* ℓ, if s directly uses ℓ, or if there exists a shape name s' such that: (1) s directly uses s' and (2) s' uses ℓ. A shapes graph (C, T) is target-restricted if, for every shape name s occurring in T, it is the case that s does not use in C any of the class and property names appearing in T. For target-restricted shapes graphs (C, T), the ASP program guarantees that repairs of the (C, T_{gr}) are also repairs of (C, T).

In the following, we assume non-recursive target-restricted shapes graphs and consider the ground versions over only node targets. For simplicity, we write node targets as shape atoms, that is we may write $T = \{s(a)\}$ instead of $T = \{(a, s)\}$. Note that if the input shapes graph is only over node targets, then it is automatically target-restricted.

4.1 Encoding into ASP

Assume a repair problem $\Psi = (G, C, T)$, where (C, T) is a non-recursive shapes graph, C is in normal form, and T is a set of node targets. 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 in databases [14], we will use special constants to annotate atoms:

- **(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:

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

P_{Repair} is a set of rules that capture violations of the constraints by the data graph (also by missing data) and propose atoms to be added or deleted so that the fixed data graph satisfies the constraints.

$P_{\text{Interpretation}}$ is a set of rules that collect all the atoms that will be in the repaired data graph. These are existing atoms from the data graph or atoms proposed to be added by the program.

$P_{\text{Constraints}}$ is a set of rules that filter out models that do not provide repairs. For instance, we add rules that prohibit the same atom to be both added and deleted by the repair program.

We are now 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. We note that T is constructed in a preliminary step to the repair program, meaning the repair additions and deletions will not change the grounding of the targets without applying the repair and recomputing.

$P_{\text{Annotation}}$

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

$$B\_(X, t^*) \leftarrow B(X) \qquad p\_(X, Y, t^*) \leftarrow p(X, Y)$$
$$B\_(X, t^*) \leftarrow B\_(X, t) \qquad p\_(X, Y, t^*) \leftarrow p\_(X, Y, t)$$

$$p\_(Y, X, t) \leftarrow p^-\_(X, Y, t) \qquad p^-\_(X, Y, t^*) \leftarrow p\_(Y, X, t^*)$$
$$p\_(Y, X, f) \leftarrow p^-\_(X, Y, f) \qquad p^-\_(X, Y, f) \leftarrow p\_(Y, X, f)$$

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 or false in the repair program, 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 (NF5) and (NF6′). We 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:

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

- 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 $\exists r.s'$, that is 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 $@new(s,X,r)$, which maps a shape name s, a node X and a property name or inverse property name r 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 r-edges from X to some Y with s' true in Y, makes one of these atoms false.

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

$P_{\text{Interpretation}}$

For every class name B, property name p and inverse property name p^- occurring in the input, we add the following rules:

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

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.

$P_{\text{Constraints}}$

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 $s \leftarrow \top$, that is of form (NF1), we add $\leftarrow s\_(Y,f)$.

(2) For each constraint of the form $s \leftarrow c$, that is 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, property name p and inverse property name p^- in the input, we add:

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

Rules (1) and (2) ensure that models preserve constraints of type (NF1) and (NF3) that 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 7.** *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.*

Proof. We need to show that for every stable model M of (G,P_Ψ), the data graph G' validates (C,T), and in particular that the updated graph G' satisfies two conditions: (i) there exists an assignment I for G' that is a model of C, and (ii) all targets in T are satisfied under I. The proof leverages the structure of the ASP program P_Ψ and its components. First, we note that G' consists

of all atoms $B(a)$ such that $B\_(a, t^{**}) \in M$ and all atoms $p(a, b)$ such that $p\_(a, b, t^{**}) \in M$; note that these atoms are confirmed to be true, that is atoms from the input data graph that were not deleted and atoms that were added by the rules. This follows from the rules in $P_{\text{Interpretation}}$ and the constraints in $P_{\text{Constraints}}$ (Rule 3), which ensure that no atom is both added and deleted.

It suffices to show that there exists an assignment I, such that for every $s(a) \in T$ it holds that $a \in [\![\varphi]\!]^I$, where $s \leftarrow \varphi$ is the constraint for s in C. Clearly, since the constraints are non-recursive, such assignment I can be extended minimally to satisfy all constraints in C for all nodes in G'.

Let M be a stable model of P_ψ. We define $I = G' \cup L$, where L contains all the shape atoms $s(a)$ such that $s\_(a, t^*) \in M$ and $s\_(a, f) \notin M$. By construction, all targets T occur in L. We show a stronger claim: for every node a and shape s with $s(a) \in I$, then $a \in [\![\varphi]\!]^I$ where $s \leftarrow \varphi \in C$. We prove this claim by induction on the depth $d(s)$ of shape s in the dependency graph. Let $s \leftarrow \varphi \in C$ be the constraint defining s in C. Recall that there is a unique constraint for each shape name in C. The depth $d(s)$ is defined recursively as follows: $d(s) = 0$ if there is no shape names appearing in φ, $d(s) = 1$ if all shape names s' appearing in φ have depth 0, that is $d(s') = 0$; and $d(s) = i$ if there exists a shape names s' in φ with $d(s') = i - 1$ and for all other shape names s'' occurring in φ it holds that $d(s'') \leq i - 1$. To prove the claim, we consider normalized constraints of form (NF1)–(NF5), (NF6′).

For the base case ($d(s) = 0$), we consider all the types of constraints with depth 0, that is s does not depend on other shape names. First, consider a constraint $s \leftarrow \top$ of the form (NF1). The rule $\leftarrow s\_(Y, f)$ added in $P_{\text{Constraints}}$ ensures no node is assigned $s\_(Y, f)$, and hence $s\_(a, f) \notin M$. Clearly, that $s\_(a, t^*) \in M$, then $a \in [\![\top]\!]^I = V(G')$ trivially. Thus, $a \in [\![\varphi]\!]_I$. Next, consider a constraint $s \leftarrow B$ of the form (NF2). By construction the following two rules are in P_Ψ: $B\_(X, t) \leftarrow s\_(X, t^*)$ and $B\_(X, f) \leftarrow s\_(X, f)$. If $s(a)$ is true in I it means that $s\_(a, t^*) \in M$ and $s\_(a, f) \notin M$. Hence, $B\_(a, t) \in M$ and $B\_(a, f) \notin M$. The fact that $B\_(a, t) \in M$ implies that $B\_(a, t^*) \in M$, which together with $B\_(a, f) \notin M$ imply that $B\_(a, t^{**}) \in M$. By construction of M, this implies that $B(a) \in G'$ and hence $a \in [\![B]\!]^I$. Finally, assume $s(a)$ was added because of a constraint of type (NF3) $s \leftarrow c$. By construction, the rules $\leftarrow s\_(X, t^*), X \neq c$ and $\leftarrow s\_(c, f)$ are in P_ψ. If $s\_(a, t^*) \in M$, then $a = c$. Thus, $a = c \in [\![c]\!]^I$.

Assume the hypothesis holds for all s' with $d(s') < d(s)$. We show the claim for constraints $s \leftarrow \varphi$ of the form (NF4), (NF5), and (NF6′). Assume a constraint of the form $s \leftarrow s_1 \wedge \cdots \wedge s_n$ (NF4). The repair rule $s_1\_(X, t^*), \ldots, s_n\_(X, t^*) \leftarrow s\_(X, t^*)$ is in P_ψ. Since M is a model, then $s\_(a, t^*) \in M$, then $s_i\_(a, t^*) \in M$ for all $1 \leq i \leq n$. By induction hypothesis ($d(s_i) < d(s)$), it holds that $a \in [\![s_i]\!]^I$ for all i. Hence, $a \in \bigcap_i [\![s_i]\!]_I = [\![\varphi]\!]^I$. Next, assume the constraint is of the form $s \leftarrow \neg s'$ (NF5). The program P_ψ contains the repair rules $s'\_(X, f) \leftarrow s\_(X, t^*)$ and $s'\_(X, t^*) \leftarrow s\_(X, f)$. Since $s\_(a, t^*) \in M$, then $s'\_(a, f) \in M$. By hypothesis it follows that $a \notin [\![s']\!]^I$, so $a \in [\![\neg s']\!]^I$. Finally, we show for constraints of the form $s \leftarrow \exists p.s'$ (NF6′). The repair rules $s'\_(@new(s, X, p), t^*)$, $p\_(X, @new(s, X, p), t) \leftarrow s\_(X, t^*)$ where Y adds a fresh node $@new(s, X, p)$ is in P_ψ. Let the fresh node for $@new(s, X, p)$ be c. By hypothesis, if $s\_(a, t^*) \in M$, then $p\_(a, c, t) \in M$ and $p\_(a, c, t^{**}) \in M$ (no deletion), so $p(a, c) \in G'$. Moreover, $s'\_(c, t^*) \in M$ implies $s'(c) \in I$ (by induction hypothesis ($d(s') < d(s)$). Hence, $a \in [\![\exists p.s']\!]^I$. ◀

We note that this theorem carries over to all the extensions, the version with cardinality constraints and constants and the version with property paths, we propose in this paper. It is easy to see that, since the rules are non-recursive in essence,[5] 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.

[5] 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.

We illustrate the repair program with a representative example.

▶ **Example 8.** Consider the repair problem $\Psi = (G, C', T)$ from Example 2, where C' is the normalized version of C. We construct the repair program P_Ψ as follows.

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

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

Intuitively, these rules together with the rules in $P_{\text{Annotation}}$ will add to the stable model the two atoms $enrolledIn\_(Ben, new_1, t^*)$ and $Course\_(new_1, t^*)$ with a fresh node new_1. For $P_{\text{Interpretation}}$, we add: $Course\_(X, t^{**}) \leftarrow Course\_(X, t^*)$, *not* $Course\_(X, f)$ for *Course_* and proceed analogously for *enrolledIn_*. For $P_{Constraints}$, we add the (constraint) rule: $\leftarrow Course\_(X, t), Course\_(X, Y, f)$ for *Course_* and proceed analogously for *enrolledIn_*. Since no atom labeled 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 $enrolledIn\_(Ben, C_1, t^{**})$, $enrolledIn\_(Ben, new_1, t^{**})$ and $Course\_(new_1, t^{**})$. The corresponding atoms $enrolledIn(Ben, C_1)$, $enrolledIn(Ben, new_1)$ and $Course(new_1)$ will form the repaired data graph G' that validates (C', T). Hence, the only repair is $(A, \emptyset)$, where A contains $enrolledIn(Ben, new_1)$ and $Course(new_1)$.

Additions and Deletions. A stable model of the repair program contains atoms annotated with "t^{**}", which represent all the atoms that have to be in the repaired data graph, i.e., it may include original atoms from the input data graph and new atoms added by the repair rules. However, 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 annotated with "t^{**}" that were not originally in the data graph, and label with *del* all the atoms annotated with "f" that were originally in the data graph. 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 and property names. The rules for inverse property names are analogous.

$$
\begin{aligned}
add(B(X)) &\leftarrow B\_(X, t^{**}), not\ B(X) & \qquad del(B(X)) &\leftarrow B\_(X, f), B(X)\\
add(p(X, Y)) &\leftarrow p\_(X, Y, t^{**}), not\ p(X, Y) & \qquad del(p(X, Y)) &\leftarrow p\_(X, Y, f), p(X, Y)
\end{aligned}
$$

Considering Example 8, the repair will label all three atoms with *add* and it will label no atoms with *del*. The repair is represented by the additions $add(enrolledIn(Ben, C_1))$, $add(enrolledIn(Ben, new_1))$ and $add(Course(new_1))$. When applying this repair to the input data graph, it will restore consistency and the resulting repaired graph will validate against the constraints.

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 already noted 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 9.** Let (G, C, T) be a repair problem, where G is empty, $T = \{s(a)\}$, and C contains the constraints: $s \leftarrow s1 \vee s2$, $s1 \leftarrow B_1$, and $s2 \leftarrow B_1 \wedge B_2$, where B_1 and B_2 are class names, and $s, s1, s2$ are shape names. To ease presentation, the constraints are not in normal form. 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 $B_1(a)$, i.e., contains $add(B_1(a))$, and the other adds both $B_1(a)$ and, the possibly redundant fact, $B_2(a)$, i.e., contains $add(B_1(a))$ and $add(B_2(a))$.

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 to 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 targets, e.g., because of conflicting constraints in shape assignments to the 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 10.** *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 some 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.

To illustrate the concept of *maximal repairs*, consider the following example.

▶ **Example 11.** Let (G, C, T) be a repair problem, where

$$G = \{\mathit{enrolledIn}(\mathit{Ben}, C_1)\}$$
$$T = \{\mathsf{StudentShape}(\mathit{Ben}), \mathsf{TeacherShape}(\mathit{Ben})\}$$
$$C = \{\mathsf{TeacherShape} \leftarrow \exists \mathit{teaches}.\mathit{Course} \wedge \neg \mathsf{StudentShape},$$
$$\mathsf{StudentShape} \leftarrow \exists \mathit{enrolledIn}.\mathit{Course}\}$$

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*. A model of the program can only include either the atom $\mathit{actualTarget}(\mathit{Ben}, \mathsf{TeacherShape})$ or the atom $\mathit{actualTarget}(\mathit{Ben}, \mathsf{StudentShape})$. The repair program chooses $\mathit{skipTarget}(\mathit{Ben}, \mathsf{TeacherShape})$ to skip the shape atom $\mathsf{TeacherShape}(\mathit{Ben})$ and repairs the shape assignment for $\mathsf{StudentShape}(\mathit{Ben})$. In this case, this is the maximum possible number of repaired shape targets. Note that skipping either $\mathsf{StudentShape}(\mathit{Ben})$ or $\mathsf{TeacherShape}(\mathit{Ben})$ will result in a repair, but only the choice to skip $\mathsf{TeacherShape}(\mathit{Ben})$ will result in a cardinality-minimal repair.

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 $s \leftarrow\!\geq_n p.s'$ with $n = 1$. We now explain the extension to support cardinality constraints with unrestricted n, which are of form $s \leftarrow\!\geq_n p.s'$. We label such constraints with (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 picking a constant is not necessary.

More precisely, for a repair problem $\Psi = (G, C, T)$, where C contains constraints of the form (NF1)-(NF5) and (NF6″), we construct the repair program P'_Ψ, which contains all the rules from P_Ψ except for the rules for (NF6′) 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: $const(c) \leftarrow$

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

$$choose(s, X, p, 0) \vee \cdots \vee choose(s, X, p, n) \leftarrow s\_(X, t^*) \tag{1}$$

$$p\_(X, @new(s, X, p, 1..i), t) \leftarrow choose(s, X, p, i), i \neq 0 \tag{2}$$

$$0\ \{p\_(X, Y, t) : const(Y)\}\ |const(Y)| \leftarrow s\_(X, t^*) \tag{3}$$

$$n\ \{s'(Y, t^*) : p\_(X, Y, t^{**})\}\ max(n, |const(Y)|) \leftarrow s\_(X, t^*) \tag{4}$$

In the following, we explain the rules (1) - (4) 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 (1) 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 (2) 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 $i \in \{1 \dots n\}$. 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 (3) to pick a number of 0 up to the maximum number of constants – using clingo's choice constructs, 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 necessary because of constraints. In addition to adding atoms over p, we need to satisfy s' on a number of n nodes. We add a rule (4) 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).

$$\ell\ \{\psi_1 \vee \psi_2\}\ \ell \leftarrow s\_(X, f), \#count\{Y : p\_(X, Y, t^*)\} = m, m > (n-1) \tag{5}$$

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 $m - (n-1)$ choices, 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 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 the 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 a 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.

▶ **Example 12.** Consider again Example 8. The repair program P'_Ψ will generate three repairs.

$$A_1 = \{Course(C_1)\}, \qquad D_1 = \{\}$$
$$A_2 = \{Course(C_1), enrolledIn(Ben, new_1)\}, \qquad D_2 = \{\}$$
$$A_3 = \{enrolledIn(Ben, new_1), Course(new_1)\}, \qquad D_3 = \{\}$$

The first repair will only add $Course(C_1)$. Intuitively, rule (1) adds $choose(s, X, p, 0)$ to the model, rule (2) and (3) will not add atoms, and rule (4) adds $s\_(C_1, t^*)$ which together with the other rules treated in Example 8 add $Course\_(C_1, t)$ and $Course\_(C_1, t^{**})$, and hence result in the repair adding $Course(C_1)$. The second repair will add $enrolledIn(Ben, new_1)$ in addition to $Course(C_1)$ because of picking $i = 1$ in rule (1), and generating a fresh value new_1 in (2), while still picking C_1 for $s\_$. The third repair will assign new_1 to $s\_$, thus resulting in the repair $(A, \emptyset)$ from Example 8. The optimization feature will return only the minimal repair that only adds $Course(C_1)$.

6 Extension with Property Path Repairs

The repair program P'_Ψ from the previous section provides repairs for shape constraints without path expressions. More precisely, it can support constraints of the form (NF1)-(NF5) and a restriction of constraints of the form $s \leftarrow\!\geq_n E.s'$, that is of form (NF6), where E can only be a property name instead of a path expression. We now show the ASP rules to support path expressions and path equality, and hence unrestricted constraints of the form (NF6) and (NF7), respectively. This completes our encoding into ASP for the SHACL fragment considered in this work.

For a repair problem $\Psi = (G, C, T)$, where C is a non-recursive set of SHACL constraints of the form (NF1)-(NF7) and T is a set of node targets, we construct a repair program P^{comp}_Ψ whose stable models provide repairs for Ψ. In particular, P^{comp}_Ψ contains all the rules from the repair program P_Ψ from Section 4 except for the rules for (NF6′) in P_{Repair}. These rules will be replaced with more general rules that capture the full constraints of the form (NF6). We will also add rules that capture constraints of the form (NF7). As in Section 5, we prioritize picking a fresh node over an existing one if picking a constant is not necessary.

▶ **Example 13.** Consider the following example. We introduce a new ReviewedPublicationShape, which verifies that a reviewed publication must have reviewers from at least three different institutions. We define this constraint using a property path with a sequence path as follows for a repair problem (G, C, T).

$$G = \{hasReviewer(Pub1, Rev1), hasReviewer(Pub1, Rev2), hasReviewer(Pub1, Rev3),$$
$$worksFor(Rev1, WUWien), worksFor(Rev2, WUWien), worksFor(Rev3, TUWien)\}$$
$$T = \{\mathsf{ReviewedPublicationShape}(Pub1)\}$$
$$C = \{\mathsf{ReviewedPublicationShape} \leftarrow\!\geq_3 hasReviewer \cdot worksFor\}$$

We assign ReviewedPublicationShape to $Pub1$. For the shape assignment to validate, we need property paths to reach at least 3 different nodes starting from node $Pub1$. However, although there are three different property paths to nodes via the reviewers $Rev1, Rev2$ and $Rev3$, only two different nodes, $WUWien$ and $TUWien$, are reachable. We need at least one additional node to be reachable via the property path to make ReviewedPublicationShape validate $Pub1$.

To repair property paths, we will first add an auxiliary rule that will be useful in the rest of the section. The idea is that for each constraint with a path expression occurring in the body of the constraint, we want to collect all the pairs of nodes that the path connects and that are relevant for the particular constraint. Specifically, for every constraint of the form $s \leftarrow \geq_n E.s'$, that is of form (NF6), or of the form $s \leftarrow E = p$, that is of form (NF7), we collect all the pairs of nodes that intuitively are the start and end of E-paths from nodes verifying s. Assume E is of the form $r_1 \cdot r_2 \cdot \cdots \cdot r_u$ and each r_i is either a property name p_i or an inverse property name p_i^-. We use a fresh auxiliary ternary predicate $p^{sE}\_$ to collect such pairs and add to P_Ψ^{comp} for each such constraint a rule:

$$p^{sE}\_(X, Y, t^*) \leftarrow s\_(X, \_), r_1\_(X, X_1, t^*), \ldots, r_u\_(X_{u-1}, Y, t^*) \tag{6}$$

For simplicity, we call instances (a, b) of $p^{sE}\_$ pairs of *source* and *end* nodes, representing the source node a verifying the shape name s and the end node b reachable from a through the path E.

Repairing constraints of form (NF6)

Assume ϕ is of the form $\geq_n E.s'$, that is the constraint for s is $s \leftarrow \geq_n E.s'$, where E is of the form $r_1 \cdot r_2 \cdot \cdots \cdot r_u$. Intuitively, for repairing the case where s is suggested to be true, that is for $s\_(X, t^*)$, we need to make sure there are E-paths to at least n nodes verifying s'. Analogously, for repairing the case where s is suggested to be false, that is for $s\_(X, f)$, we need to make sure that less than n nodes are reachable from X through E-paths. We now present the rules for each case, which will be added in the P_{Repair} part of the program.

- We describe the rules for the case when $s\_(X, t^*)$. We first add the following rules:

$$choose(s, X, p^{sE}, 0) \vee \cdots \vee choose(s, X, p^{sE}, n) \leftarrow s\_(X, t^*) \tag{7}$$
$$p^{sE}\_(X, @new(s, X, p^{sE}, 1..i), t) \leftarrow choose(s, X, p^{sE}, i), i \neq 0 \tag{8}$$
$$0\ \{p^{sE}\_(X, Y, t) : const(Y)\}\ |const(Y)| \leftarrow s\_(X, t^*) \tag{9}$$
$$n\ \{s'\_(Y, t^*) : p^{sE}\_(X, Y, t^{**})\}\ max(n, |const(Y)|) \leftarrow s\_(X, t^*) \tag{10}$$

To make sure we consider all E-paths in G and do not generate more values than needed, we make use of the auxiliary predicate $p^{sE}\_$. We add the rules (7)-(10) analogously to the rules (1)-(4) from the previous section, which generate just enough fresh values Y as end nodes in $p^{sE}\_$ to satisfy the cardinality n. Similarly, we pick a number up to the maximum number of constants for $p^{sE}\_$ and we pick a number of n end nodes in $p^{sE}\_$ to satisfy s'. For constant reuse optimization, we add the $\#minimize\{1@1, X, Y : p^{sE}\_(X, Y, t), const(Y)\}$ optimization rule to P_Ψ^{comp}.

We add rules, which for each property r_i at some node X_v suggest to either generate a fresh value to be r_i-connected to X_v or keep some existing r_i atom. We first add the case for generating r_1 separately and then for each $1 \leq i \leq u - 2$, we add the following rules:

$$choose(s, X, r_1, 0) \vee choose(s, X, r_1, 1) \leftarrow p^{sE}\_(X, Y, t) \tag{11}$$
$$r_1\_(X, @new(s, X, r_1, 1), t) \leftarrow choose(s, X, r_1, 1) \tag{12}$$
$$choose(s, X_i, r_{i+1}, 0) \vee choose(s, X_i, r_{i+1}, 1) \leftarrow p^{sE}\_(X, Y, t), r_1\_(X, X_1, t^{**}), \ldots, r_i\_(X_{i-1}, X_i, t^{**}) \tag{13}$$
$$r_{i+1}\_(X_i, @new(s, X_i, r_{i+1}, 1), t) \leftarrow choose(s, X_i, r_{i+1}, 1), r_1\_(X, X_1, t^{**}), \ldots, r_i\_(X_{i-1}, X_i, t^{**}) \tag{14}$$

Note that in the above rules X_0 stands for X and X_u stands for Y. We have generated the paths from nodes verifying s up to r_{u-1}. The following rule allows to generate last edges r_u from X_{u-1} to some Y that can either be a constant generated by rule (9), or a fresh value generated by rule (8). This rule together with rule (10) and rule (6) will make sure there are enough E-paths connected to enough values to satisfy the cardinality n. We add the rule:

$$0\ \{r_u\_(X_{u-1}, Y, t)\}\ 1 \leftarrow p^{sE}\_(X, Y, t), r_1\_(X, X_1, t^{**}), \ldots, r_{u-1}\_(X_{u-2}, X_{u-1}, t^{**}) \tag{15}$$

- For the case where a property shape s is suggested to be false at X with a cardinality of n, i.e., for $s\_(X, f)$, we pick from all atoms $p^{sE}\_(X, Y, t^*)$ to either cut the E-path or falsify s' at Y. We add the following rules:

$$\ell\ \{\psi_3 \vee \psi_4\}\ \ell \leftarrow s\_(X, f), \#count\{Y : p^{sE}\_(X, Y, t^*)\} = m, m > (n-1) \tag{16}$$

where $\ell = m - (n-1)$, ψ_3 is the expression $p^{sE}\_(X, Y, f) : p^{sE}\_(X, Y, t^*), not\ s'\_(Y, f)$ and ψ_4 is the expression $s'\_(Y, f) : p^{sE}\_(X, Y, t^*), not\ p^{sE}\_(X, Y, f)$.

$$r_1\_(X, X_1, f) \vee \cdots \vee r_u\_(X_{u-1}, Y, f) \leftarrow p^{sE}\_(X, Y, f), r_1\_(X, X_1, t^*), \ldots, r_u\_(X_{u-1}, Y, t^*) \tag{17}$$

We add rule (16) analogous to rule (5) to pick from all E-paths $p^{sE}\_(X, Y, t^*)$. To make $s\_$ false at X, we have two disjunctive options that we can falsify. The first option is to cut the E-path by falsifying one of the atoms $r_1 \cdot r_2 \cdot \cdots \cdot r_u$. 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 E-path from start node X to end node Y was not falsified via $p^{sE}\_(X, Y, f)$ atom. Again, by picking $m-(n-1)$ choices, we make sure that only the maximum allowed cardinality will be in the repaired graph. Consequently, we add rule (17) to cut paths, if an E-path was suggested to be false via $p^{sE}\_(X, Y, f)$ atom, by picking from all E-path atoms $r_1 \cdot r_2 \cdot \cdots \cdot r_u$ (exactly) one atom to falsify, thereby cutting the E-path. Note that multiple (potentially overlapping) cuts for different E-paths will be minimized regarding cardinality by the optimization rules.

Repairing constraints of form (NF7)

Finally, assume ϕ is of the form $E = p$, where E is a path of the form $r_1 \cdot r_2 \cdot \cdots \cdot r_u$ and p is a property name. Intuitively, for repairing the case where s is suggested to be true, that is for $s\_(X, t^*)$, we need to make sure there is a property atom $p(X, Y)$ for every E-path from X to Y and vice versa. Analogously, for repairing the case where s is suggested to be false, that is for $s\_(X, f)$, we need to make sure that for at least one E-path from X to Y there is no property atom $p(X, Y)$ or vice versa. We now present the rules for each case, which will be added in the P_{Repair} part of the program.

- For the case where s is suggested to be true at $p^{sE}\_(X, Y, t^*)$ with equality to a property $p(X, Y)$, we add the following rules to make sure that for every E-path from X to Y there is also a property atom $p(X, Y)$ in the repair. We repair this case by using a disjunctive rule to either add or delete property atoms.

$$p\_(X, Y, t) \vee p^{sE}\_(X, Y, f) \leftarrow s\_(X, t^*), p^{sE}\_(X, Y, t^*) \tag{18}$$

$$p^{sE}\_(X, Y, t) \vee p\_(X, Y, f) \leftarrow s\_(X, t^*), p\_(X, Y, t^*) \tag{19}$$

The above rules will make sure to generate properties and auxiliary properties to ensure the equality. Rule (18) suggests for every E-path from X to Y that is suggested to be true to either make true $p(X, Y)$ or to make false the E-path. Rule (19) does the same for every $p(X, Y)$ suggested to be true.

Now the auxiliary properties together with rules (11) to (15) will propose actual paths from the source to the end nodes generated in the auxiliary properties. Since these rules may or may not construct the paths, we add constraints (25) and (26), which together with interpretation rule (23) will enforce the generation of the entire path.

- For the case where s is suggested to be false, we add rules which make sure that there is at least one $p^{sE}\_(X,Y,t^*)$ with no $p\_(X,Y,t^*)$ or one $p\_(X,Y,t^*)$ with no $p^{sE}\_(X,Y,t^*)$. Again, we can repair this case by using a disjunctive rule to either add or delete property atoms.

$$notEquals(s,X,p^{sE},p) \leftarrow s\_(X,f), p^{sE}\_(X,Y,t^*), not\ p\_(X,Y,t^*) \tag{20}$$

$$notEquals(s,X,p^{sE},p) \leftarrow s\_(X,f), not\ p^{sE}\_(X,Y,t^*), p\_(X,Y,t^*) \tag{21}$$

$$\begin{aligned} 1\ \{&p^{sE}\_(X,Y,f) : p\_(X,Y,t^*) \vee p\_(X,Y,f) : p^{sE}\_(X,Y,t^*) \vee \\ &p^{sE}\_(X, @new(s,X,p^{sE},1), t) \vee p\_(X, @new(s,X,p,1), t)\}\ 1 \\ &\leftarrow s\_(X,f), not\ notEquals(s,X,p^{sE},p) \end{aligned} \tag{22}$$

We add two rules (20) and (21), which use a fresh *notEquals* predicate, to collect cases where there is a $p^{sE}\_(X,Y,t^*)$ with no $p\_(X,Y,t^*)$ or there is a $p\_(X,Y,t^*)$ with no $p^{sE}\_(X,Y,t^*)$. If no such a case exists, then the repair rule (22) creates exactly one such case by choosing one of four disjunctive options. Either it suggests $p^{sE}\_(X,Y,f)$ where there is a $p\_(X,Y,t^*)$, suggests $p\_(X,Y,f)$ where there is a $p^{sE}\_(X,Y,t^*)$, suggests a fresh $p^{sE}\_(X, @new(s,X,p^{sE},1), t)$ or suggests a fresh $p\_(X, @new(s,X,p,1), t)$. Each of these options will falsify the equality for $s\_(X,f)$.

$P_{\text{Interpretation}}$

We add an interpretation rule which makes $p^{sE}\_$ true for the repair if all property atoms on the path $r_1 \cdot r_2 \cdot \cdots \cdot r_u$ are true in the repair.

$$\begin{aligned} &p^{sE}\_(X,Y,t^{**}) \\ &\qquad \leftarrow p^{sE}\_(X,Y,t^*), not\ p^{sE}\_(X,Y,f), r_1\_(X,X_1,t^{**}), \ldots, r_u\_(X_{u-1},Y,t^{**}) \end{aligned} \tag{23}$$

$P_{\text{Constraints}}$

We add a constraint rule for $p^{sE}\_$ to filter models that are not repairs.

$$\leftarrow p^{sE}\_(X,Y,t^*), p^{sE}\_(X,Y,f) \tag{24}$$

We also add constraints to filter models which do not repair an equality.

$$\leftarrow s\_(X,t^*), p^{sE}\_(X,Y,t^{**}), not\ p\_(X,Y,t^{**}) \tag{25}$$

$$\leftarrow s\_(X,t^*), p\_(X,Y,t^{**}), not\ p^{sE}\_(X,Y,t^{**}) \tag{26}$$

The above rules conclude our encoding into ASP of the SHACL fragment considered in this paper. The correctness of the encoding is captured in the following theorem.

▶ **Theorem 14.** *Assume a repair problem $\Psi = (G,C,T)$, where C is a non-recursive set of SHACL constraints of the form (NF1)-(NF7) and T is a set of node targets. We construct a repair program P_Ψ^{comp} where, for every stable model M of P_Ψ^{comp}, (A,D) is a maximal repair for Ψ, where (A,D) is obtained as follows:*

- *A contains all atoms of the form $B(a)$, $p(a,b)$ not occurring in G such that $B\_(a,t^{**})$ and $p\_(a,b,t^{**})$ are in M, and*
- *D contains all atoms of the form $B(a)$, $p(a,b)$ from G such that $B\_(a,f)$ and $p\_(a,b,f)$ in M.*

Idea. Let $\Psi = (G, C, T)$ be a repair problem where C is a non-recursive set of SHACL constraints in normal form (NF1)–(NF7) and T is a set of node targets. Let P_Ψ^{comp} be the ASP program constructed in Section 6. Let M be an arbitrary stable model of of $(G, P_\Psi^{\text{comp}})$, let (A, D) be the repair from M obtained as above, and let G' be the repaired data graph obtained as follows $G' = (G \cup A) \setminus D$. First, that (A, D) is a maximal repair for Ψ by Definition 10 means that 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. Intuitively, this is ensured by the soft constraints, which allow to skip node targets if a stable model for all the node target is not found. Let $T^* \subseteq T$ be the set of node targets $s(a)$ from T such that $s\_(a, t^*) \in M$. Then, by construction, $actualTarget(a, s) \in M$ for each such node target $s(a) \in T^*$, which together with the optimization rule that minimizes the *skipTarget* atoms and M being a stable model, implies that there is no node target $s'(a') \in T \setminus T^*$ that could be repaired.

Finally, we show that (A, D) is indeed a repair for (G, C, T^*). Let I be a shape assignment defined as $I = G' \cup L$, where $L = \{s(a) \mid s\_(a, t^*) \in M \text{ and } s\_(a, f) \notin M\}$. Arguing analogously to the proof of Theorem 7, it suffices to show that for every node a and shape s with $s(a) \in I$, then $a \in [\![\varphi]\!]^I$ where $s \leftarrow \varphi \in C$. The shape assignment I can be naturally extended to a model of all the constraints by simply evaluating the bodies of the constraints starting from shapes with depth 0, updating I by adding the corresponding shape atoms, and then proceeding with depth 1 and so on until no new shape atoms are entailed. Let I' be the new shape assignment obtained this way. Clearly, I' satisfies all the constraints and includes the targets.

The proof is analogous to the proof of Theorem 7, that is by induction on the depth $d(s)$ of each shape name s in the dependency graph of C. We show the claim (the induction step) only for the additional constraints of the form (NF6) that are supported in P_Ψ^{comp}. The base case for constraints of the form (NF7) uses similar arguments.

Suppose $s(a) \in I$ and the constraint for s in C is $s \leftarrow \geq_n E.s'$ of the form (NF6). We need to show that $a \in [\![\geq_n E.s']\!]^I$, that is there exist at least n distinct nodes Y such that $(a, Y) \in [\![E]\!]_I$ and $s'(Y) \in I$. By construction of I, it must be that $s\_(a, t^*) \in M$ and $s\_(a, f) \notin M$.

From rule (7), the presence of $s\_(a, t^*) \in M$ implies that M contains exactly one atom of the form $choose(s, a, p^{sE}, i)$ for some $i \in \{0, \ldots, n\}$. If $i > 0$, then rule (8) ensures the presence of atoms $p^{sE}\_(a, @new(s, a, p^{sE}, j), t)$ in M for $j = 1, \ldots, i$ (i.e., i fresh nodes are created). In addition, rule (9) allows the selection of existing constants Y_k such that $p^{sE}\_(a, Y_k, t) \in M$. Rule (10) then enforces that at least n such nodes Y satisfy $s'\_(Y, t^*) \in M$.

Now consider each such Y for which $p^{sE}\_(a, Y, t^{**}) \in M$. Rule (23) guarantees that $(a, Y) \in [\![E]\!]^I$: it ensures that $p^{sE}\_(a, Y, t^*) \in M$, $p^{sE}\_(a, Y, f) \notin M$, and all atoms representing the path $r_1\_(a, X_1, t^{**}), \ldots, r_u\_(X_{u-1}, Y, t^{**})$ are present in M. Additionally, rules (11)–(15) construct the actual path edges to validate this.

Since the depth of s' is strictly less than that of s, we apply the induction hypothesis: for all Y with $s'\_(Y, t^*) \in M$, it follows that $s'(Y) \in I$, that is $Y \in [\![s']\!]^I$.

The nodes Y are distinct by construction since fresh nodes are uniquely generated via $@new$, and constants are inherently distinct. Finally, rule (10) ensures that there are at least n such distinct Y satisfying both $(a, Y) \in [\![E]\!]^I$ and $Y \in [\![s']\!]^I$. Hence, $a \in [\![\geq n\, E.s']\!]^I$, as required. ◀

In the following, we illustrate the property path repair program with examples.

▶ **Example 15.** We revisit Example 13 and change the constraints for ReviewedPpublicationShape to not only validate for three different institutions, but also to validate that at least three different reviewers review a publication. We define these two constraints using property paths, where one path is a sequence path. To illustrate the repair for inverse paths, we exchange the *hasReviewer*

property with a *reviews* property, which navigates backwards from reviewer to publication. We also change G to include two reviewers for $Pub1$ with two different institutions $WUWien$ and $TUWien$.

$$G = \{hasReviewer(Pub1, Rev1), hasReviewer(Pub1, Rev2), \\ worksFor(Rev1, WUWien), worksFor(Rev2, TUWien)\}$$
$$T = \{\mathsf{ReviewedPublicationShape}(Pub1)\}$$
$$C = \{\mathsf{ReviewedPublicationShape} \leftarrow \geq_3 reviews^- \wedge \geq_3 reviews^- \cdot worksFor\}$$

The ReviewedPublicationShape contains two property constraints. The first constraint is a minimum cardinality of three on the (inverse) path $reviews^-$, which indicate a reviewer for a publication. The second constraint is again a minimum cardinality of three on the (sequence) path $reviews^- \cdot worksFor$, which indicate that there have to be at least three institutions that the reviewers work for. With these two constraints we express that there need to be at least three reviewers from at least three different institutions for a reviewed publication. We represent the path for $reviews^-$ with an auxiliary property atom ${p_1^s}^E\_$ and the path for $reviews^- \cdot worksFor$ with an auxiliary property atom ${p_2^s}^E\_$.

We assign ReviewedPublicationShape to $Pub1$. However, it does not validate. There are only two reviewers $Rev1$ and $Rev2$ with institutions $WUWien$ and $TUWien$, respectively. To repair the shape assignment, the repair program picks three ${p_1^s}^E\_$ atoms to be made true for the path $reviews^-$ and three ${p_2^s}^E\_$ atoms to be made true for the path $reviews^- \cdot worksFor$ via rule (10). To satisfy this rule, it generates one fresh value for each of ${p_1^s}^E\_$ and ${p_2^s}^E\_$ via rules (7) and (8). As a further consequence, the repair program adds fresh values for property atoms $reviews^-$ and $worksFor$ via rules (11), (12) and (15), thereby generating the paths to satisfy both cardinality constraints of $\geq_3$. We get the following minimal repairs.

$$A_1 = \{reviews(new_1, Pub1), worksFor(Rev1, new_2)\}, \qquad D_1 = \{\}$$
$$A_2 = \{reviews(new_1, Pub1), worksFor(Rev2, new_2)\}, \qquad D_2 = \{\}$$
$$A_3 = \{reviews(new_1, Pub1), worksFor(new_1, new_2)\}, \qquad D_3 = \{\}$$

The minimal repairs fix the cardinality constraints by adding a property atom $reviews(new_1, Pub1)$ with a fresh node new_1 and by adding a property atom *reviews* with a fresh node new_2 to either $Rev1$, $Rev2$ or new_1. The result is a data graph with three end nodes for $supports^-$ and three end nodes for $supports^- \cdot worksFor$, which validates the shape assignment for ReviewedPublicationShape.

▶ **Example 16.** Consider the following example. We introduce a new CourseLimitShape to represent a limitation of the number of participants for a course. Also, we want to keep the registered participants in sync with the IDs of the registered students.

$$G = \{enrolledIn(Ann, Course1), enrolledIn(Ben, Course1), enrolledIn(Bob, Course1), \\ hasStudentID(Ann, 2119110), hasStudentID(Ben, 1716110), hasStudentID(Bob, 9427084), \\ participantID(Course1, 2119110)\}$$
$$T = \{\mathsf{CourseLimitShape}(Course1)\}$$
$$C = \{\mathsf{CourseLimitShape} \leftarrow \leq_2 enrolledIn^- \cdot hasStudentID \\ \wedge\ enrolledIn^- \cdot hasStudentID = participantID\}$$

The CourseLimitShape, which indicates that a course can only have a maximum of two participants, is assigned to course $Course1$. The constraint is checked by means of the (sequence) path $enrolledIn^- \cdot hasStudentID$, which uses an inverse property $enrolledIn^-$ to navigate backwards

from a course to a student. Also, a course gets the student IDs of the participants noted via the property *participantID* which has to be equal to the student IDs of students enrolled in the course via the path $enrolledIn^- \cdot hasStudentID$.

We assign CourseLimitShape to $Course1$. However, it does not validate, because there are too many students enrolled in $Course1$ (Ann, Ben and Bob). Also the equality constraint on *participantID* is not satisfied for student ID nodes 1716110 and 9427084. To repair the shape assignment, the repair program picks one $p^{sE}\_$ atom (3 existing $p^{sE}\_$ atoms minus a maximum cardinality of 2) to be made false via rule (16), and as a further consequence picks from atoms on the path $enrolledIn^- \cdot hasStudentID$ to be made false via rule (17), thereby cutting the path and satisfying the cardinality constraint of $\leq_2$. Also, the remaining two paths need to satisfy the equality on *participantID* via rules (18) and (19). We get the following minimal repairs.

$$A_1 = \{participantID(Course1, 9427084)\}, \quad D_1 = \{enrolledIn(Ben, Course1)\}$$
$$A_2 = \{participantID(Course1, 9427084)\}, \quad D_2 = \{hasStudentID(Ben, 1716110)\}$$
$$A_3 = \{participantID(Course1, 1716110)\}, \quad D_3 = \{enrolledIn(Bob, Course1)\}$$
$$A_4 = \{participantID(Course1, 1716110)\}, \quad D_4 = \{hasStudentID(Bob, 9427084)\}$$

The minimal repairs fix the cardinality constraint by either removing Ben or removing Bob from $Course1$ by cutting the path via false advise on either *enrolledIn* or *hasStudentID*, while at the same time adding the *participantID* of the remaining node (9427084 or 1716110) via true advise. The result is a data graph with two $Course1$ participants with their student IDs also noted via *participantID* for $Course1$, which validates the shape assignment for CourseLimitShape.

7 ASP Implementation

We developed a prototypical system for implementing SHACL repair programs using the Java programming language and the ASP system clingo.[6] The prototype parses a SHACL shapes graph and a data graph, both in Turtle syntax, and translates 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 addition triples and deletion triples as repairs for the data graph. The implementation is available online on github.[7]

Generating SHACL repair programs

In the following, we explain the process of running SHACL repairs in detail. Figure 1 shows the processing flow through the individual processing steps.

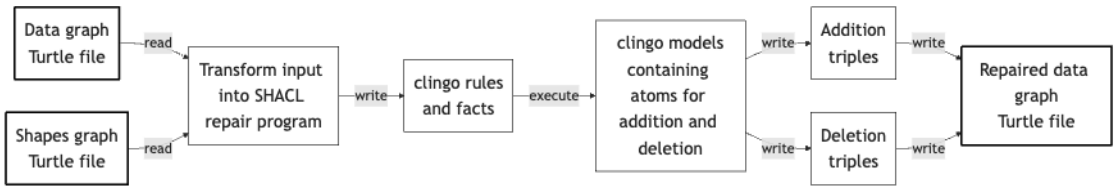


Figure 1 SHACL repair process.

[6] https://potassco.org/clingo/
[7] https://github.com/robert-david/shacl-repairs

- First, input data, which are Turtle files containing the data graph and the shapes graph, are read by the Java program.
- The Java program processes the input and translates it into a SHACL repair program consisting of clingo rules and facts.
- clingo then runs the repair program and returns the answer sets (stable models) containing atoms for additions and deletions.
- The Java program reads the answer sets and writes addition and deletion triples in Turtle syntax.
- The Java program uses the addition and deletion triples to change the input data graph and create a repaired data graph, which is then written into a file in Turtle syntax.

Translating SHACL shape constraints from RDF syntax to abstract syntax

For implementing the translation, we first need to translate from SHACL shapes in RDF Turtle syntax into our abstract syntax, which is then further translated into ASP to generate the repair program. In the following, we show the translation of SHACL shape constraints from Turtle syntax into our abstract syntax. For some of the constraints, like class membership using *sh:class*, there is a simple translation into an equivalent shape expression in abstract syntax. For other constraints, like minimum cardinality constraint *sh:minCount*, we need to translate into a complex shape expression.

Table 1 shows the list of translations from SHACL constraint components in RDF syntax defined in [32] to complex shape expressions in abstract syntax.

▶ **Example 17.** We present an example to illustrate how this translation is done for a SHACL shape consisting of several constraints. The shape *:StudentShape* below is translated into a shape expression as described in Section 2.

```
:StudentShape a sh:NodeShape;
    sh:targetNode :Ben;
    sh:property [
        sh:path :enrolledIn;
        sh:minCount 1;
        sh:maxCount 1;
        sh:or (
            [ sh:class :Course ]
            [ sh:in ("ID1" "ID2") ]
        )
    ] .
```

The translation into a shape expression is as follows. We introduce new shape names for conjunctions (*sh:and*) and disjunctions (*sh:or*) for better readability. Note that SHACL constraints on the same node or property are interpreted as a conjunction as stated in [32].

$: StudentShape \leftarrow s_1 \wedge s_2$

$s_1 \leftarrow \geq_1 : enrolledIn.s_3 \wedge \neg \geq_1 : enrolledIn.\neg s_3$

$s_2 \leftarrow \neg \geq_2 : enrolledIn.s_4 \wedge \neg \geq_1 : enrolledIn.\neg s_3$

$s_3 \leftarrow s_5 \vee s_6$

$s_5 \leftarrow : Course$

$s_6 \leftarrow \neg(\neg"ID1" \wedge \neg"ID2") \wedge \neg(\neg"ID1" \wedge \vee"ID2")$

This shape expression of *StudentShape* can be translated into normal form and consequently into ASP rules as explained in Section 5.

Table 1 SHACL Constraints Translation.

SHACL constraint component	Example SHACL constraint in RDF	Complex shape expression in abstract syntax
Cardinality constraint components		
sh:minCount	[sh:path E ; sh:minCount 2]	$s \leftarrow \geq_2 E.\top$
sh:maxCount	[sh:path E ; sh:maxCount 2]	$s \leftarrow \neg \geq_3 E.\top$
Logical constraint components		
sh:and	sh:and (:A :B)	$s \leftarrow A \wedge B$
sh:or	sh:or (:A :B)	$s \leftarrow A \vee B$
sh:not	sh:not :A	$s \leftarrow \neg A$
sh:xone	sh:xone (:A :B)	$s \leftarrow (A \wedge \neg B) \vee (\neg A \wedge B)$
Shape-based constraint components		
sh:qualifiedMinCount	[sh:path E ; sh:qualifiedMinCount 2 ; sh:qualifiedValueShape :A]	$s \leftarrow \geq_2 E.A$
sh:qualifiedMaxCount	[sh:path E ; sh:qualifiedMaxCount 2 ; sh:qualifiedValueShape :A]	$s \leftarrow \neg \geq_3 E.A$
Other constraint components		
sh:hasValue	[sh:path E ; sh:hasValue :c]	$s \leftarrow \exists E.c$
sh:in	[sh:path E ; sh:in (:c1 :c2)]	$s \leftarrow \exists E.(c_1 \vee c_2)$

8 Tests for SHACL repair programs

To validate our implementation of the SHACL repair program, we decided to test against the official SHACL data shapes test suite, *SHACL Test Suite and Implementation Report*.[8] We then consider a real-world scenario over Wikidata as a data graph and test the performance against SHACL shapes with Wikidata constraints [26].

The test suite is intended for SHACL processors to test against the test cases and identify how far they correctly cover the SHACL W3C recommendation with their implementation. It also provides a list of published test results from implementations that have submitted their test results. The tests are maintained by members of the Working Group. In our view, this is the best candidate to test the SHACL repair program, because the test suite is officially approved and maintained by the SHACL authors, it provides a sufficient coverage regarding the SHACL W3C recommendation for testing validation and it is publicly available. Besides the SHACL data shapes test suite, we also developed unit tests as part of our implementation, which are intended to provide a sufficient coverage for the implemented SHACL repairs.

For testing performance in a real-world scenario, we use sample data from the Wikidata KG and SHACL shapes intended to check Wikidata constraints. We select a fixed set of shapes and scale the sample size to get insights about the performance of our approach with respect to the size of the data graph.

In the following, we first describe the unit tests that were defined as part of the implementation. Then, we go into the details of the SHACL data shapes test suite and describe which test cases were repaired and which gaps are still there regarding our implementation. We then present our test set of SHACL shapes for Wikidata constraints, describe how we create the data graph samples and discuss the performance test results.

Unit Test Suite

To verify the repair program implementation, we created a unit test suite that covers all the implemented shape constraints. The idea is to have minimal independent examples that the repair implementation can test against. We grouped these examples in seven groups for class constraints, property constraints, property path constraints, equality on property paths, value (constant) constraints, logical constraints and constraints that will cause a conflict either within a shape or between multiple shape assignments. The seven groups also include logical expressions for conjunction, disjunction and negation, while the (dedicated) logical constraints tests cover the *sh:xone* constraint component. The unit test suite consists of a total of 91 test cases.

Data Shapes Test Suite

The main target of our tests is the SHACL data shapes test suite. We looked into the test suite to identify a maximal subset of test cases that is covered by our current implementation. We identified a subset of 33 test cases from the 121 test cases in the test suite. We ran the SHACL repair program against the selected test cases and validated the results by comparing the resulting repairs with the validation results provided as part of the test cases. Although the validation results do not directly show the repair options (as described in [3]), they provide insights to determine if the resulting repairs are viable. In the following, we provide the details of the repair test results for the selected 33 test cases of the SHACL data shapes test suite. We group them into 3 groups, which are *node*, *path* and *property*, as named to indicate the purpose in the SHACL

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

Table 2 SHACL Unit Tests.

Unit Test Suite	
Constraint Test Group	Nr of Tests
Class	8
Property	9
Property path	35
Equals	6
Value (constant)	19
Logical (sh:xone)	4
Conflict	10

data shapes test suite. All test cases were repaired successfully, if a repair is possible. For 30 test cases, the repairs resulted in a data graph that does not cause any violations on SHACL validation. For 3 test cases there are skipped targets and a full repair of the data graph is not possible regarding cardinality-minimal repairs as defined in this paper. The test scenarios showed that our implementation works according to the defined SHACL repair program to repair the test cases.

Table 3 SHACL Test Results – Node constraints.

SHACL data shapes test suite – node			
Test Name	Nr of Repair Models	Ok/repaired/skip Targets	Comments
and-001	1	1/2/0	sh:and
and-002	2	1/2/0	sh:and
class-001	1	2/2/0	sh:class
class-002	1	2/2/0	sh:class
class-003	1	2/4/0	sh:class with multiple classes, overlapping target sets
hasValue-001	1	1/0/1	sh:hasValue
in-001	1	3/0/1	sh:in
node-001	1	1/1/0	sh:node
not-001	1	1/1/0	sh:not
not-002	1	1/1/0	sh:not
or-001	2	2/3/0	sh:or
xone-001	3	2/1/0	sh:xone
xone-duplicate	1	0/0/2	validation report for shape xone-duplicate

Table 4 SHACL Test Results – Path constraints.

SHACL data shapes test suite – path			
Test Name	Nr of Repair Models	Ok/repaired/skip Targets	Comments
path-complex-002	1	0/2/0	path sequence and sh:inversePath
path-inverse-001	3	1/2/0	sh:inversePath
path-sequence-001	1	2/2/0	path sequence
path-sequence-002	1	2/2/0	path sequence
path-strange-001	3	2/1/0	two valid paths together
path-strange-002	3	2/1/0	valid and invalid paths together

Table 5 SHACL Test Results – Property constraints.

SHACL data shapes test suite – property			
Test Name	Nr of Repair Models	Ok/repaired/skip Targets	Comments
and-001	5	1/3/0	sh:and
class-001	4	2/1/0	sh:class
equals-001	1	2/4/0	sh:equals
hasValue-001	1	2/1/0	sh:hasValue
in-001	1	2/1/0	sh:in
maxCount-001	2	1/1/0	sh:maxCount
maxCount-002	1	1/1/0	sh:maxCount
minCount-001	1	1/1/0	sh:minCount
minCount-002	1	1/0/0	sh:minCount
node-001	1	3/1/0	sh:node
node-002	2	1/1/0	sh:node
not-001	2	2/1/0	sh:not
or-001	3	2/1/0	sh:or
qualifiedValueShape-001	1	0/1/0	sh:qualifiedValueShape

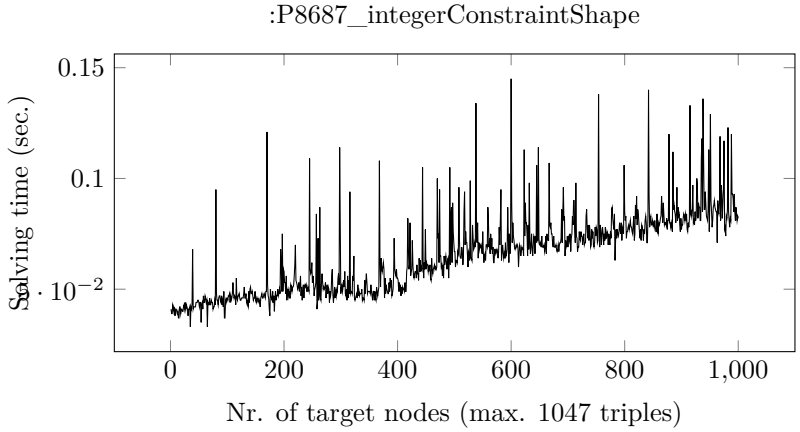
Table 6 SHACL Unit Tests.

Wikidata SHACL Shapes		
Nr	Shape Name	Constraint Components and Paths
1	P8687_integerConstraintShape	sh:datatype
2	:P26_SymmetricShape	sh:inversePath, sh:equals, sh:minCount
3	:P1083_NoBoundsShape	sequence path, sh:and, sh:maxCount
4	:P1469_ItemRequiresStatementShape	sh:in, sh:qualifiedValueShape, sh:qualifiedMinCount
5	:P1283_ValueRequiresStatementShapes	sh:or, sequence path, sh:in, sh:qualifiedValueShape, sh:qualifiedMinCount

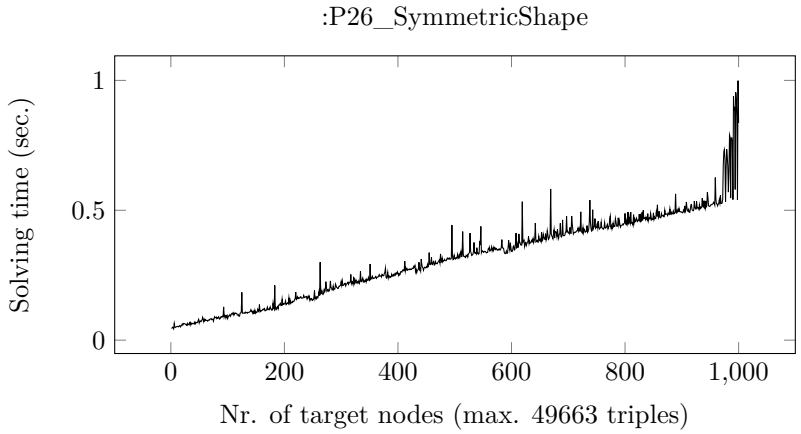
Wikidata Performance Tests

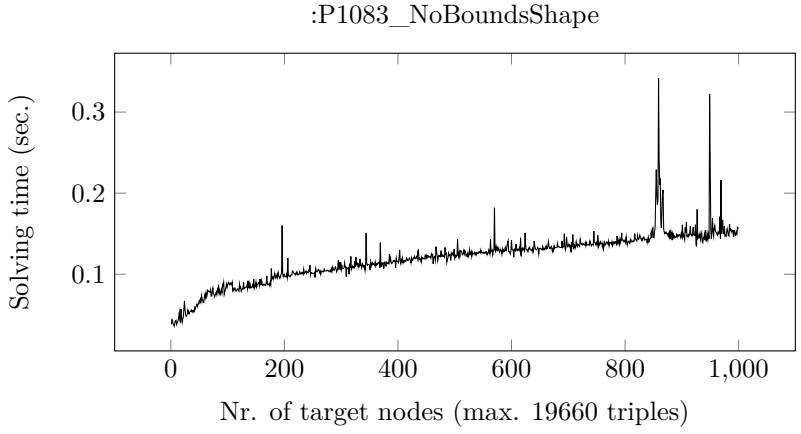
Wikidata [39] is a collaborative, free and open knowledge base providing structured data from Wikimedia projects, like Wikipedia.[9] Wikidata provides constraint types to validate data, which are defined by the community to ensure data quality for collaborative editing. However, Wikidata constraints are not enforced and violations in the data can be found.

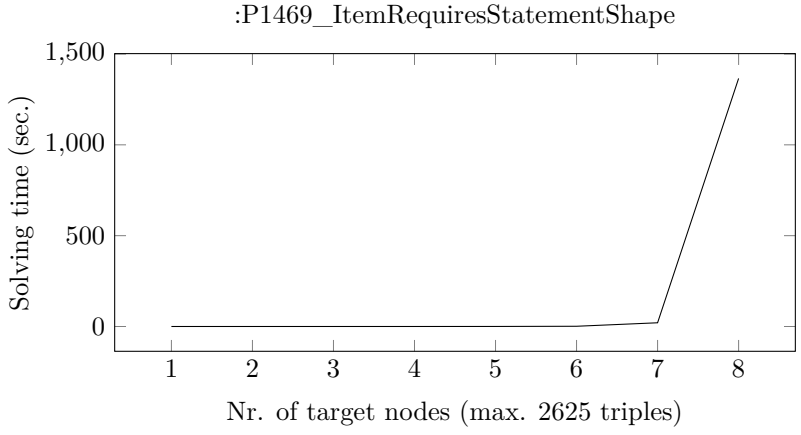
Recent work on Wikidata constraint checking by Ferranti et al. [26] provided publicly available SHACL shapes,[10] which formalize Wikidata constraints to be validated by SHACL processors. For testing the performance of SHACL repair programs, we selected 5 representative Wikidata shapes to be used to repair Wikidata RDF graph samples. The shapes (1 to 5) have an increasing number of constraints and cover the supported shape expressions. Table 6 shows the details of the selected shapes and constraints. We then retrieved RDF graph samples from Wikidata via SPARQL endpoint, where we specifically retrieved seven data sets, which cause violations regarding the seven shapes. Performance tests scale up the number of shape target nodes from 1 to a maximum of 1000. We also limited the processing time of running the repair programs in clingo to 100 minutes of time. The Wikidata performance diagrams (below) show the time to compute repairs in relation to the number of target nodes. We also note the maximum size of the (sample) data graph, which is the number of triples for the maximum number of target nodes tested.

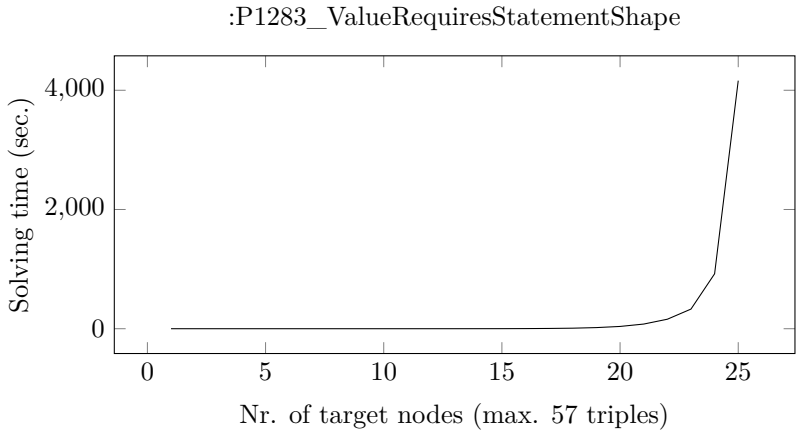


[9] https://www.wikidata.org/
[10] https://github.com/nicolasferranti/wikidata-constraints-formalization/tree/main/constraints-formalization


:P26_SymmetricShape
Solving time (sec.)
1
0.5
0
0
200
400
600
800
1,000
Nr. of target nodes (max. 49663 triples)


:P1083_NoBoundsShape
Solving time (sec.)
0.3
0.2
0.1
0
200
400
600
800
1,000
Nr. of target nodes (max. 19660 triples)


:P1469_ItemRequiresStatementShape
Solving time (sec.)
1,500
1,000
500
0
1
2
3
4
5
6
7
8
Nr. of target nodes (max. 2625 triples)



For the tests of shapes 1,2 and 3, we can see a linear behavior when scaling the number of target nodes. Generally, the performance scales well and we are able to solve the maximum tested target nodes within hundreds of milliseconds. However, the tests for shapes 4 and 5 show an above linear behavior and we run into performance problems already with a low number of target nodes (9 for shape 4 and 25 for shape 5). Closer evaluation showed that the reason for this behavior is the use of constant constraints (using the *sh:in* constraint) and the constant reuse optimization.

Generally, with the exception of constants constraints, our approach shows good scalability regarding the size of the data graph. Further tests with large scale data graphs are required to gain further insights into the limitations.

9 Conclusion

We presented an approach to minimally repairing a data graph so that it conforms to a set of SHACL constraints. We first analyze the types of repair that may be desirable in practice. Specifically, we observe that to repair cardinality constraints, it may not be desirable to reuse existing nodes from the graph, as this could introduce incorrect facts into the data. Therefore, we consider two scenarios: one where repairs always introduce fresh values to satisfy cardinality constraints, and another where repairs may reuse existing nodes if necessary, with a preference for using fresh nodes.

Inspired by existing work in databases, we propose Answer Set Programming encodings of the repair problem that account for these different repair scenarios. We first describe an encoding for a restricted setting, which forces the program to introduce fresh values to satisfy existential constraints. We then extend this to a setting that handles arbitrary cardinality, allowing for the reuse of existing constants when necessary. Additionally, we provide an encoding to support the repair of property path constraints and equality constraints.

In case not all the shape targets can be repaired, we optimize to repair as many of them as possible. We developed a prototypical system for implementing SHACL repair programs using the Java and the ASP system clingo. With the repair program and the ASP implementation, we lay the foundation for bringing repairs into practical scenarios, and thus improving the quality of RDF graphs in practice. We tested the implementation with unit tests, the data shapes test suite and performance tests on Wikidata as a real-world data set, which show promising results, but also some limitations, regarding the applicability in practice.

Future work. Several tasks remain for future work. For the practical side, the next step will be to select further use cases where we can apply the repairs and evaluate the practical feasibility and explore repair quality and scalability. Notably, our SHACL repair approach has already been applied in practice to resolve ambiguities in ontology RDF graphs [24]. An important next step is to also add support for repair preferences. The notion of minimal repairs considered in this paper does not take into account the possible user preferences regarding additions or deletions of concrete facts. E.g., a user might prioritize deletions of facts that are older or are coming from less reliable sources. Such setting has led, e.g., to the notions of global, Pareto and completion optimality of repairs [38]. Adapting these and other notions to SHACL is an important task for future work. This will help with decision making for users to select a repair and thereby improve the usability when applying repairs in practice.

Another important direction is to support *recursive* SHACL constraints, and hence, also *unrestricted* class-based and property-based targets. This is 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. However, it might be important to not rule out relevant practical use cases.

References

1 Shqiponja Ahmetaj, Iovka Boneva, Jan Hidders, Katja Hose, Maxime Jakubowski, José Emilio Labra Gayo, Wim Martens, F. Mogavero, Filip Murlak, Cem Okulmus, Axel Polleres, Ognjen Savkovic, Mantas Simkus, and Dominik Tomaszuk. Common foundations for shacl, shex, and pg-schema. *The Web Conference (WWW)*, 2025. To appear. doi:10.48550/arXiv.2502.01295.

2 Shqiponja Ahmetaj, Diego Calvanese, Magdalena Ortiz, and Mantas Šimkus. Managing change in graph-structured data using description logics. In *AAAI-14*. AAAI Press, 2014. doi:10.48550/arXiv.1404.4274.

3 Shqiponja Ahmetaj, Robert David, Magdalena Ortiz, Axel Polleres, Bojken Shehu, and Mantas Simkus. Reasoning about explanations for non-validation in SHACL. In Meghyn Bienvenu, Gerhard Lakemeyer, and Esra Erdem, editors, *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*, pages 12–21, 2021. doi:10.24963/kr.2021/2.

4 Shqiponja Ahmetaj, Robert David, Axel Polleres, and Mantas Simkus. Repairing SHACL constraint violations using answer set programming. In *The Semantic Web - ISWC 2022 - 21st International Semantic Web Conference, Virtual Event, October 23-27, 2022, Proceedings*, volume 13489 of *Lecture Notes in Computer Science*, pages 375–391. Springer, 2022. doi:10.1007/978-3-031-19433-7_22.

5 Shqiponja Ahmetaj, Timo Camillo Merkl, and Reinhard Pichler. Consistent query answering over SHACL constraints. In Pierre Marquis, Magdalena Ortiz, and Maurice Pagnucco, editors, *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR 2024, Hanoi, Vietnam. November 2-8, 2024*, 2024. doi:10.24963/KR.2024/1.

6 Medina Andresel, Julien Corman, Magdalena Ortiz, Juan L. Reutter, Ognjen Savkovic, and Mantas Šimkus. Stable model semantics for recursive SHACL. In *Proc. of The Web Conference 2020*, WWW '20, pages 1570–1580. ACM, 2020. doi:10.1145/3366423.3380229.

7 Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proc. of PODS*, pages 68–79. ACM Press, 1999. doi:10.1145/303976.303983.

8 Franz Baader. Optimal Repairs in Ontology Engineering as Pseudo-Contractions in Belief Change. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, pages 983–990, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3555776.3577719.

9 Franz Baader. Relating Optimal Repairs in Ontology Engineering with Contraction Operations in Belief Change. *SIGAPP Appl. Comput. Rev.*, 23(3):5–18, September 2023. doi:10.1145/3626307.3626308.

10 Franz Baader, Patrick Koopmann, Francesco Kriegel, and Adrian Nuradiansyah. Computing Optimal Repairs of Quantified ABoxes w.r.t. Static EL TBoxes. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, pages 309–326, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-79876-5_18.

11 Franz Baader, Patrick Koopmann, Francesco Kriegel, and Adrian Nuradiansyah. Optimal ABox Repair w.r.t. Static EL TBoxes: From Quantified ABoxes Back To ABoxes. In *The Semantic Web: 19th International Conference, ESWC 2022, Hersonissos, Crete, Greece, May 29 – June 2, 2022, Proceedings*, pages 130–146, Berlin, Heidelberg, 2022. Springer-Verlag. doi:10.1007/978-3-031-06981-9_8.

12 Franz Baader and Francesco Kriegel. Pushing Optimal ABox Repair from EL Towards More Expressive Horn-DLs. In *Proceedings of the 19th*

International Conference on Principles of Knowledge Representation and Reasoning, pages 22–32, August 2022. doi:10.24963/kr.2022/3.

13 Franz Baader and Renata Wassermann. Contractions Based on Optimal Repairs. *LTCS-Report 24-03*, 2024.

14 Leopoldo E. Bertossi. *Database Repairing and Consistent Query Answering*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011. doi:10.2200/S00379ED1V01Y201108DTM020.

15 Meghyn Bienvenu, Camille Bourgaux, and François Goasdoué. Querying inconsistent description logic knowledge bases under preferred repair semantics. In *AAAI*. AAAI Press, 2014. doi:10.1609/aaai.v28i1.8855.

16 Meghyn Bienvenu, Camille Bourgaux, and François Goasdoué. Explaining inconsistency-tolerant query answering over description logic knowledge bases. In *AAAI*. AAAI Press, 2016. doi:10.1609/aaai.v30i1.10092.

17 Meghyn Bienvenu and Riccardo Rosati. Tractable approximations of consistent query answering for robust ontology-based data access. In Francesca Rossi, editor, *IJCAI*. IJCAI/AAAI, 2013. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6904.

18 Diego Calvanese, Davide Lanti, Ana Ozaki, Rafael Peñaloza, and Guohui Xiao. Enriching ontology-based data access with provenance. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*. AAAI Press, 2019.

19 Diego Calvanese, Magdalena Ortiz, and Mantas Šimkus. Verification of Evolving Graph-structured Data under Expressive Path Constraints. In *ICDT*, 2016. doi:10.4230/LIPIcs.ICDT.2016.15.

20 Diego Calvanese, Magdalena Ortiz, Mantas Simkus, and Giorgio Stefanoni. Reasoning about explanations for negative query answers in DL-Lite. *J. Artif. Intell. Res.*, 48:635–669, 2013. doi:10.1613/jair.3870.

21 İsmail İlkan Ceylan, Thomas Lukasiewicz, Enrico Malizia, Cristian Molinaro, and Andrius Vaicenavicius. Explanations for negative query answers under existential rules. In *Proc. of KR 2020*, pages 223–232, 2020. doi:10.24963/kr.2020/23.

22 Julien Corman, Juan L. Reutter, and Ognjen Savkovic. Semantics and validation of recursive SHACL. In *Proc. of ISWC'18*. Springer, 2018. doi:10.1007/978-3-030-00671-6_19.

23 Robert David. SHACL Repair Program Implementation. Software, version 1.2.1., swhId: swh:1:dir:9de47c8b97cba079add751f9e2a64421238a67f3 (visited on 2025-12-08). URL: https://github.com/robert-david/shacl-repairs, doi:10.4230/artifacts.25261.

24 Robert David, Albin Ahmeti, Shqiponja Ahmetaj, and Axel Polleres. OWLstrict: A Constrained OWL Fragment to avoid Ambiguities for Knowledge Graph Practitioners. In *The Semantic Web: 22nd European Semantic Web Conference, ESWC 2025, Portoroz, Slovenia, June 1–5, 2025, Proceedings, Part II*, pages 47–64, Berlin, Heidelberg, 2025. Springer-Verlag. doi:10.1007/978-3-031-94578-6_3.

25 Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutiérrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009. doi:10.1007/978-3-642-03754-2_2.

26 Nicolas Ferranti, Axel Polleres, Jairo Francisco de Souza, and Shqiponja Ahmetaj. Formalizing property constraints in wikidata. In *Wikidata@ ISWC*, 2022.

27 Mikhail Galkin, Sören Auer, Maria-Esther Vidal, and Simon Scerri. Enterprise knowledge graphs: A semantic approach for knowledge management in the next generation of enterprise information systems. In *ICEIS (2)*, pages 88–98, 2017. doi:10.5220/0006325200880098.

28 José Emilio Labra Gayo, Eric Prud'hommeaux, Iovka Boneva, and Dimitris Kontokostas. *Validating RDF Data*. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool Publishers, 2017. doi:10.2200/S00786ED1V01Y201707WBE016.

29 Maertin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019. doi:10.1017/S1471068418000054.

30 Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutiérrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. *Knowledge Graphs*. Synthesis Lectures on Data, Semantics, and Knowledge. Morgan & Claypool Publishers, 2021. doi:10.2200/S01125ED1V01Y202109DSK022.

31 Aditya Kalyanpur, Bijan Parsia, Matthew Horridge, and Evren Sirin. Finding all justifications of OWL DL entailments. In *The Semantic Web: 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007+ ASWC 2007, Busan, Korea, November 11-15, 2007. Proceedings*, pages 267–280. Springer, 2007. doi:10.1007/978-3-540-76298-0_20.

32 Holger Knublauch and Dimitris Kontokostas. Shapes constraint language (SHACL). Technical report, W3C, July 2017. URL: https://www.w3.org/TR/shacl/.

33 Domenico Lembo, Maurizio Lenzerini, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. Inconsistency-tolerant semantics for description logics. In *RR*, Lecture Notes in Computer Science. Springer, 2010. doi:10.1007/978-3-642-15918-3_9.

34 Ying Li and Patrick Lambrix. Repairing $\mathcal{EL}$ ontologies using weakening and completing. In *European Semantic Web Conference*, pages 298–315. Springer, 2023. doi:10.1007/978-3-031-33455-9_18.

35 Mónica Caniupán Marileo and Leopoldo E. Bertossi. The consistency extractor system: Answer

set programs for consistent query answering in databases. *Data Knowl. Eng.*, 69:545–572, 2010. doi:10.1016/j.datak.2010.01.005.

36 A. Ozaki and Rafael Peñaloza. Provenance in ontology-based data access. *Description Logics*, 2018. URL: https://api.semanticscholar.org/CorpusID:53179150.

37 Rafael Peñaloza. Axiom pinpointing. In *Applications and Practices in Ontology Design, Extraction, and Reasoning*, pages 162–177. IOS Press, 2020. doi:10.3233/SSW200042.

38 Slawek Staworko, Jan Chomicki, and Jerzy Marcinkowski. Prioritized repairing and consistent query answering in relational databases. *Ann. Math. Artif. Intell.*, 64(2-3):209–246, 2012. doi:10.1007/s10472-012-9288-8.

39 Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014. doi:10.1145/2629489.