A two-layered Approach to Cope with Recursion in SHACL Repairs

Robert David[1,2]

[1]Vienna University of Economics and Business, Vienna, Austria
[2]Semantic Web Company doing business as Graphwise, Vienna, Austria

Abstract

The Shapes Constraint Language SHACL allows to define recursive shapes, i.e., a shape may refer to itself, either directly or via other shape references. However, SHACL does not define a semantics for such recursions and the W3C recommendation leaves the interpretation to implementers to decide. A similar problem is encountered when repairing SHACL constraint violations. Changing a data graph to conform to given SHACL constraints might change the shape targets. For example, by fixing a class constraint by adding a class (membership) to a node on the graph, one changes the target nodes of shapes defining the same class as target class. This potentially introduces recursion into repairs and might lead to infinite generation of new nodes. Implementers are left to decide how to cope with such a situation. In this paper, which discusses the SHACL repair recursion problem in the context of recent work on SHACL repairs, we look at strategies for repair implementations to identify and mitigate infinite node generation via target recursion. We propose a combination of (existing) approaches to provide a practical two-layer strategy. With this work, SHACL users can take advantage of target-recursive repairs while avoiding infinite recursion and thereby support important real-world scenarios.

Keywords

SHACL, Constraints, Data repairs, Recursion, Answer Set Programming

1. Introduction

The Shapes Constraint Language (SHACL)[1] [1] is the W3C recommendation to validate RDF data in a data graph against constraints, which are grouped into so-called shapes. Shapes can refer to other shapes in constraints, and therefore SHACL allows to define recursive shapes. Although this is possible at the syntactic level, the semantics of validating such recursive shapes is not defined in the W3C recommendation. Implementers are left with making their own choices. A similar problem can be observed when repairing, i.e., modifying a data graph to conform to given SHACL constraints, because even when not considering recursive shapes, a recursive cycle can be introduced via shape targets if the repair is allowed to change target class membership of nodes and property use in the data graph. This can result in infinite new nodes (triples) being generated, which is a problem for implementations which compute such repairs and which need to find some way to handle this situation.

This work builds upon SHACL repairs as described in [2], which computes repairs as sets of additions and deletions to modify a data graph to conform to SHACL constraints. SHACL allows different options to determine target nodes for shapes. Class and property targets implicitly select nodes that are class members or that are subjects or objects of property triples. When class membership or property use changes in the data graph because of a data change to repair a constraint violation, the set of shape target nodes changes as well if the specific class or property was used for shape targeting. As a consequence, a recursive cycle is introduced via shape targets, which might result in further repair changes, further target nodes, and potentially produces infinite new triples. The following example illustrates the situation of changing shape target nodes.

QKG@ESWC 2026 — Evaluating, Improving, and Sustaining Knowledge Graph Quality. Co-located with the Extended Semantic Web Conference, May 10 – 14, 2026, Dubrovnik, Croatia

*Corresponding author.
robert.david@graphwise.ai (R. David)
https://graphwise.ai/ (R. David)
0000-0002-3244-5341 (R. David)

[1]https://www.w3.org/TR/shacl/

Example 1. Assume a shapes graph with shape constraints for StudentShape and CourseShape

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

```
:CourseShape a sh:NodeShape;
    sh:targetClass :Course;
    sh:property [
        sh:path :courseID;
        sh:minCount 1;
        sh:maxCount 1; ] .
```

and a data graph

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

The shapes graph states that each node conforming to StudentShape must be enrolled in at least one course. It also states that each node conforming to CourseShape must have exactly one $courseID$. While StudentShape is verified at node Ben, CourseShape determines the shape targets via class membership with $Course$. The data graph graph states that Ben is enrolled in $C1$.

Validation checks whether there is at least one (*sh:minCount*) outgoing edge $enrolledIn$ from target node Ben to a node and that the node is an instance of the $Course$ class. It does not check any nodes for CourseShape, because there are not any members of $Course$ in the data graph. Validation fails, because $C1$ is not a class member of $Course$.

Repairing the data graph would add $C1$ to $Course$ so that it validates. However, by doing so, $C1$ becomes an implicit target node for CourseShape.

In the publication defining the SHACL repair program [2], this scenario is explicitly excluded and the target nodes are grounded before running the actual repair process. However, in practice this might rule out important use cases.

In this paper, we add support for target recursion and discuss different ways to avoid infinite target recursion when repairing SHACL constraint violations by introducing guards. We i) define cases where infinite recursion can actually happen to identify them prior to repair program execution, and, if identified, ii) introduce guards into the repair program to prevent non-termination. This strategy aims to allow for a maximum of use case coverage, i.e., we only stop execution if an actual infinite recursion would occur.

This paper is structured as follows. In Section 2, we discuss related work and background work that this paper builds upon. In Section 3, we describe the *repair target recursion* problem in detail and identify the problematic repair scenarios where infinite recursion happens. We then propose a two-layered approach to identify and handle these scenarios to avoid non-termination. In Section 4, we describe how repair target recursion and guards against infinite target recursion can be implemented as an extension of the SHACL repair program. Finally, we close with conclusions and next steps in Section 5.

2. Background and Related Work

2.1. SHACL Validation

In this paper, we use the abstract syntax defined in [2] for RDF and SHACL in a restricted form.

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 *shape expression* ϕ is of the form:

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

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

where $s \in N_S$, $B \in N_C$, $c \in N_N$, n is a positive integer and $p \in N_P$. 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 (S, T), where S 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 S *recursive* if there is a shape name s that directly or indirectly refers to itself in the shapes constraints. In this paper, we assume non-recursive constraints.

2.2. Recursive Inference

Recursion is a basic method to solve computational problems by using functions that call themselves. Important to recursion is to understand how to define a base case for the function, which acts as a stopping condition to terminate the execution eventually. If such a case is not well defined, infinite recursion and non-termination of the program occurs. Properly understanding and handling of recursion in programs is therefore of major interest [3].

2.3. SHACL recursion

Recursion in SHACL is mainly discussed for the case of mutual cyclic dependencies of shapes. The SHACL W3C recommendation allows recursion structurally but does not define a semantics. This has lead to research being done in this regard and several publications address this challenge [4, 5, 6, 7]. Recursion is introduced because of the graph-based structure of shapes, which define graph patterns to evaluate via constraints, which also can refer to further shapes.

2.4. SHACL repairs

Recent work introduced a novel approach to repair SHACL constraint violations by modifying the data to conform to the defined constraints [2]. SHACL repairs is implemented as a repair program that, given SHACL constraints, can compute repairs as cardinality-minimal changes to the data graph to achieve conformance. The repairs are represented as sets of additions and deletions [8] of triples, which are computed by the repair program for a set of SHACL shapes and an input data graph. Adding the additions to the data and removing the deletions from the data will result in a new data graph that is consistent with the shape constraints. The implementation of the repair program is a combination of a Java program, which reads the SHACL shapes and the data graph and generates an answer set program (ASP) [9], and the clingo [10] solver, which processes the program to determine the repairs. ASP is a declarative problem-solving paradigm based on logic programming and nonmonotonic reasoning. It allows us to represent a problem as a nonmonotonic logic program and use an ASP solver to compute its solutions. The solutions, so-called answer sets or stable models, are minimal models of a logic program. With answer set programming, we can even resolve potentially conflicting constraints, and programs will eventually stabilize into minimal models providing repairs. SHACL repairs optimizes to return minimal models that i) fix a maximum of violations and ii) do a cardinality-minimal number of changes

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

(additions and deletions). SHACL repairs took a design decision for fixing minimum cardinality constraints by adding new triples always with fresh values and not reuse existing values from the data graph.

Example 2. Consider again Example 1 with StudentShape and CourseShape. We update the data graph to add some incorrect data as follows:

```
:Ben :enrolledIn :C1 .
:C2 a :Course ;
    :courseID "1", "2" .
```

The SHACL repair program will propose two different minimal models to repair the data graph to satisfy the shape constraints. These models contain the repair tuples (additions, deletions) (A_1, D_1) and (A_2, D_2), respectively.

$$A_1 = \{Course(C1)\} \qquad D_1 = \{courseID(C2, \text{"1"})\}$$
$$A_2 = \{Course(C1)\} \qquad D_2 = \{courseID(C2, \text{"2"})\}$$

Applying either repair to the data graph will result in conformance of the initial target nodes Ben and $C2$ with the constraints of StudentShape and CourseShape, respectively. Note that without target recursion, the repair program will not assign CourseShape to $C1$, which results in non-conformance of the repaired data graph with the constraints.

3. Methodology

In Example 1, we showed how new target nodes might be introduced when repairing graph data to conform to SHACL constraints. This example is not problematic regarding changing target nodes, because there is no recursive cycle via shape targets. In the following, we first define *repair target recursion*, i.e., shape definitions where a recursive cycle is introduced via targets and constraints. We then define basic scenarios for SHACL shapes, where infinite repair target recursion occurs. We follow with a discussion how these cases can avoid infinite recursion and propose a two-layered approach to guard against it.

3.1. Repair target recursion

Proposition 1. *A shapes graph (S, T) is* repair target recursive *if the shapes names in the constraints S use one or more of the class and property names appearing in the targets T.*

Changes done to the data graph to satisfy constraints involving class and property names appearing in T will modify the set of target nodes. If validation fails on one or more of these class and property constraints for a data graph G, then the set of target nodes $T = \{s(a)\}$ is changed via a repair (A, D) being applied to G.

Note that repair target recursion was excluded in [2]. Targets were grounded according to (W, s) to determine target nodes before repairing the data graph, i.e., repair additions or deletions to the data graph were not considered to change the set of target nodes. A recomputing of the grounding after the repair is required to consider changed target nodes.

3.2. Basic scenarios for infinite recursion

Repair target recursion may lead to a change of target nodes. However, it does not necessarily lead to infinite recursion. A repair tuple (A, D) modifies the shape target nodes if a class used for *class target* is added or removed from a node in the data graph or if a property atom used for *subjects-of target* or *objects-of target* is added or removed from the data graph. Clearly, removing target nodes does not create a scenario for infinite recursion, because the repair is limited by the (structure of) the data graph, i.e., a finite number of initial target nodes, and the recursion ends at latest with an empty set of target nodes.

Adding target nodes, however, might result in an infinite recursion. This happens either as a result of repairing a minimum cardinality constraint or via a constant constraint requiring some specific value. Note that the SHACL repair program, as defined in [2], generates fresh values when adding property atoms to satisfy cardinality constraints. It does not reuse existing values from the data graph.

In the following, we define basic scenarios where infinite recursion can occur. We then discuss how these scenarios can be identified to differentiate them from other recursion scenarios and thereby still allow repairs to normally compute cases without infinite recursion. Finally, we show how we can mitigate infinite recursion by introducing guard options to prevent non-termination and still get meaningful repair results.

Identifying basic scenarios for infinite recursion The following basic scenarios result in an infinite recursion.

- If there is a repair target recursion via a *class target* and there is a property constraint with a minimum cardinality of at least 1 (via *sh:minCount or sh:qualifiedMinCount*) or there is a constant constraint (via *sh:hasValue or sh:in*) with the target class being a constraint on the value node, then there is an infinite target recursion.

  Definition 1. *Let* (W, s) *be a target with* W *a* class target *for class* B *and either* s *is a shape expression of the form* $s \leftarrow\!\!\geq_i p.B, i \geq 1$ *or* s *is a shape expression of the form* $s \leftarrow\!\!\geq_i p.(c \wedge B), i \geq 0$. (W, s) *is a* class target recursion R_C.

- If there is a recursive cycle via a *subjects-of target* or *objects-of target*, meaning there is a SHACL minimum cardinality property constraint (via *sh:minCount or sh:qualifiedMinCount*) or there is a constant constraint (via *sh:hasValue or sh:in*), then there is an infinite target recursion.

  Definition 2. *Let* (W, s) *be a target with* W *a* subjects-of target *for property* p *and either* s *is a shape expression of the form* $s \leftarrow\!\!\geq_i p.s', i \geq 1$ *or* s *is a shape expression of the form* $s \leftarrow\!\!\geq_i p.c, i \geq 0$. (W, s) *is a* subjects-of target recursion R_{PS}. *Likewise for* objects-of target *and* (W, s) *is a* objects-of target recursion R_{PO}.

Note that these scenarios depend on the data graph as well. For example, if there are no target nodes involved in recursive cycles in the first place, then infinite recursion does not occur in this specific scenario.

Besides these basic scenarios, there are extended scenarios to be considered. Property paths with cardinality constraints might as well lead to infinite recursion. Disjunction of constraints, on the other hand, might avoid infinite recursion with at least one finite alternative. In this paper, however, we limit the scenarios to the two basic ones defined above and leave investigation of further scenarios to future work.

3.3. Guarding against infinite recursion

Based on the basic scenarios, we introduce a two-layered approach to guard against infinite recursion while still allowing for non-infinite recursive cases. Generally, there are different approaches discussed in the literature for preventing non-termination because of infinite cycles. Some, but not all of them, apply to our scenarios. We distinguish prevention based on an analysis of the shapes graph structure (static cycle detection) from guard approaches which are introduced into the computation (repair program guards). The two-layered approach is intended to work automatically as an extension of the SHACL repair implementation.

3.3.1. Layer 1: static cycle detection

Cycle detection statically checks the RDF shapes graph if it contains some recursive cycles that match the two basic scenarios defined above. It can be easily done by creating a dependency graph for the target classes and properties and their usage in constraints across all shapes in the shapes graph. The

basic scenarios can then be checked via the RDF triples. If such a scenario is identified in the shapes graph, and there are (initial) target nodes from the data graph involved in a recursive cycle, then the layer 2 measures will be automatically introduced into the repair program to ensure termination. Otherwise, the repair computation can proceed safely as normal.

In this paper, we only consider the basic scenarios. However, static cycle detection can be straightforwardly expanded to also consider further SHACL constraints.

3.3.2. Layer 2: repair program guards

Layer 2 introduces guards into the repair program, if layer 1 identified a potential infinite cycle. Guards against non-termination because of infinite cycles can take several forms. However, they are dependent on the underlying implementation. For example, if the implementation does not allow for lazy evaluation (e.g., as in the Haskell programming language), the infinite structure will always be fully evaluated. In the following, we present some general approaches to avoid infinite recursion.

- Value fixpoint: a measured value is increased or decreased towards a base case fixpoint. Recursion stops when this fixpoint is reached.
- Graph structure: recursion processes on increasingly smaller parts of the graph and stops when a minimum base case is reached.
- Explicit depth limit: recursion is counted and stops when a predefined count is reached.
- Explicit resource limit: the underlying processor running the program measures some resource and hard-stops the computation when a predefined limit is reached. For example, stopping the computation after 1 minute of processing.

Given our specific repair problem and the implementation (and limitations of) using ASP, we decided to restrict recursion to process on increasingly smaller parts of the facts. Specifically, we restrict fresh values for added property atoms to be (exclusively) picked from a predefined set of facts (finite domain). Recursion will stop when no values are left to pick from. The implementation details are discussed in the next section.

4. Implementation

In the following, we first add support for repair target recursion to the repair program. We then discuss how the two-layered approach can be implemented for ASP. The layer 1 static cycle detection is independent of the repair implementation and can be done in a preliminary step prior to the actual repair program execution. The layer 2 is dependent on the repair implementation and we discuss how guards can be implemented in ASP to avoid non-termination because of infinite target recursion.

4.1. Implementing target recursion

The implementation of the SHACL repair program is based on ASP. In the following, we provide a simplified version of the repair rules used by the program to discuss how the set of target nodes can be changed by repairs. For each atom $s(a) \in T$, we add repair rules depending on the constraint ϕ. The body captures the shape targets and the head proposes repairs, i.e., changes to the data graph to satisfy a given constraint. For simplicity, we only show the rule for $\exists p.s'$, i.e., $\geq_1 p.s'$.
- If ϕ is a class name B, then we add the rule:

$$B(X) \leftarrow s(X)$$

- If ϕ is of the form $\exists p.s'$, then we add a new p-edge from X to a fresh node (generated unique value via function *@new()*) and assign the node to s'.

$$s'(@new()), p(X, @new()) \leftarrow s(X)$$

Changing the shape targeting The original version of the repair program grounds the target nodes first and then runs the repair. We replace these rules with new rules to, instead of grounding, allow for a rule-based target selection:

$$R_C : s(X) \leftarrow B(X)$$
$$R_{PS} : s(X) \leftarrow p(X, Y)$$
$$R_{PO} : s(Y) \leftarrow p(X, Y)$$

Example 3. Assume a shapes graph with shape constraints for StudentShape and a (modified) CourseShape

```
:StudentShape a sh:NodeShape;
    sh:targetNode :Ben;
    sh:targetClass :Student;
    sh:property [
        sh:path :enrolledIn;
        sh:minCount 1;
        sh:class :Course; ] .
```

```
:CourseShape a sh:NodeShape ;
    sh:targetClass :Course;
    sh:property [
        sh:path :enrolledStudent;
        sh:minCount 1;
        sh:class :Student; ] .
```

and a data graph

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

The modified StudentShape adds $Student$ as a target class. The modified CourseShape states that there must be at least one (*sh:minCount*) outgoing edge $enrolledStudent$ and the value node must be an instance of $Student$.

This example introduces an infinite target recursion via the classes $Student$ and $Course$. Both classes are used for targeting and the *sh:class* constraint occurs in both shapes mutually.

Repairing the data graph would add $Course$ to $C1$ so that it validates. $C1$ then becomes an implicit target node for CourseShape, which in turn adds a new edge for $enrolledStudent$ with a fresh value. This value is required to be an instance of $Student$ and requires another $enrolledIn$ edge to be added. This recursive cycle continues infinitely.

In our example, based on the data graph, the repair includes the following additions:

$$A = \{Course(C1), enrolledStudent(C1, new_1), Student(new_1),$$
$$enrolledIn(new_1, new_2), Course(new_2), \ \ldots\}$$

The set of additions is infinite, which in practice can only be represented in a lazy fashion. However, to repair the data graph, the actual complete set must be applied. This means it is not possible to apply the repair in practice. Next, we discuss a solution for the ASP implementation to guard against such a scenario, preventing infinite target recursion and thereby non-termination.

4.2. Implementing a target recursion guard

In ASP, guards are arithmetic comparison conditions to restrict variable values and are placed inside rule bodies. The comparison eventually prevents the rule body from being satisfied and thereby the rule head from being concluded.

In the repair target recursion case, the rules of the repair program allow for an unbounded term growth for new property atoms, because the strategy is to always generate a fresh value. We want to limit this generation by introducing a guard comparing the value count or by some counter providing a depth limit. In ASP, aggregates can be used to represent a count of an increasing number of fresh values that are generated. However, aggregates are applied only in the solving step after the grounding step. In our case, the infinite generation of fresh values prevents the program from being grounded.

A solution approach is to restrict the fresh values used for added property atoms to a finite domain, represented as facts of the program, and not dynamically generate them as part of the solving. This ensures the grounding step terminates. In the following, we show the modified rules which implement a finite domain for fresh values. The size of the domain can be user-configured.

- To introduce a finite domain for fresh values, we add to the repair program a number of facts n according to the (user) configuration of the domain size. We add the following facts.

$$new(k), 1 \leq k \leq n$$

This adds n *new* atoms to the program to be used as fresh values in property atom generation.
- We modify the rule for $\exists p.s'$ as defined above. If ϕ is of the form $\exists p.s'$, then we add a new p-edge from X to a *new* node and assign the node to s'.

$$0\ \{s'(Y), p(X,Y)\}\ 1 \leftarrow s(X), new(Y)$$

The rule uses clingo's choice construct $\{\ldots\}$ to pick a subset of 0 or 1 of the listed atoms to be true in order to satisfy the existential constraint $\exists p.s'$. For Example 3, we would generate the following rules.

$$0\ \{Course(Y), enrolledIn(X,Y)\}\ 1 \leftarrow s(X), new(Y)$$
$$0\ \{Student(Y), enrolledStudent(X,Y)\}\ 1 \leftarrow s(X), new(Y)$$

Furthermore, we need to ensure that values are not reused (as intended in the original implementation by fresh values). To achieve this, we add constraints for each pair of properties involved in the identified basic scenarios. In our case, these are the properties $enrolledIn$ and $enrolledStudent$.

$$\leftarrow enrolledIn(X,Y), enrolledStudent(Z,Y), new(Y)$$
$$\leftarrow enrolledIn(X,Y), enrolledIn(Z,Y), new(Y), X \neq Z$$
$$\leftarrow enrolledStudent(X,Y), enrolledStudent(Z,Y), new(Y), X \neq Z$$

Example 4. Revisiting Example 3, we do not get a repair anymore, but a termination of the repair program reporting a skipped target for $C1$. The shape assignment $\mathsf{CourseShape}(C1)$ would validate by using a *new* value, but eventually the recursion uses up all (unused) values from the finite domain and the constraints prevent the repair. Therefore this shape assignment is not repaired and reported as skipped as described in [2].

We note that the modified rules for this case only need to be used, i.e., the repair program to be modified, in the case of layer 1 reporting an actual infinite recursion basic scenario. However, the shapes and data graph may include infinite and finite cases, where some might be repaired (e.g., Example 1) while others are not, as long as the finite domain provides a sufficient number of fresh values to be used for adding property atoms.

A prototypical implementation for the target recursion guard for class targets, including the 3 examples, is available on github.[4]

5. Conclusions

In this paper, we present an approach for SHACL repairs to implement repair target recursion for class- and property-based shape targets. We discuss a two-layered approach to prevent infinite recursion from happening for two basic scenarios. Furthermore, we show how guards can be implemented for ASP repair programs.

Future work will address the following open questions. This initial work can be expanded from the two basic scenarios to the full SHACL definition of [2], and specifically address property paths and disjunction. Although the presented approach can also be applied similarly to these scenarios, there might be more optimal solutions to be investigated. Also, further guards can be determined to prevent non-termination for the ASP implementation. A limitation of the presented approach is the predetermined domain size, which might be difficult to decide for in practice. Adding further configuration options (e.g., different domain sizes for each property) can help to better adjust to different use cases. Finally, future work applies the two-layered approach to real-world use cases to determine the practical value. Previously identified use cases, such as time series data [11], benefit from infinite target recursion guards when performing data repairs.

[4] https://github.com/robert-david/shacl-repairs/tree/target-recursion-guard

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] H. Knublauch, D. Kontokostas, Shapes constraint language (SHACL), Technical Report, W3C, 2017. URL: https://www.w3.org/TR/shacl/.

[2] S. Ahmetaj, R. David, A. Polleres, M. Šimkus, A Logic Programming Approach to Repairing SHACL Constraint Violations, Transactions on Graph Data and Knowledge 3 (2025) 1:1–1:36. URL: https://drops.dagstuhl.de/entities/document/10.4230/TGDK.3.3.1. doi:10.4230/TGDK.3.3.1.

[3] D. E. Smith, M. R. Genesereth, M. L. Ginsberg, Controlling recursive inference, Artificial Intelligence 30 (1986) 343–389.

[4] M. Andresel, J. Corman, M. Ortiz, J. L. Reutter, O. Savkovic, M. Simkus, Stable model semantics for recursive shacl, in: Proceedings of The Web Conference 2020, 2020, pp. 1570–1580.

[5] B. Bogaerts, M. Jakubowski, Fixpoint semantics for recursive shacl, arXiv preprint arXiv:2109.08285 (2021).

[6] C. Lahaye, Towards efficient validation of rdf graphs against recursive shacl, 2020.

[7] J. Corman, F. Florenzano, J. L. Reutter, O. Savkovic, Shacl2sparql: Validating a sparql endpoint against recursive shacl constraints., in: ISWC (Satellites), 2019, pp. 165–168.

[8] S. Ahmetaj, R. David, M. Ortiz, A. Polleres, B. Shehu, M. Simkus, Reasoning about Explanations for Non-validation in SHACL, in: M. Bienvenu, G. Lakemeyer, E. Erdem (Eds.), Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021, 2021, pp. 12–21. doi:10.24963/kr.2021/2.

[9] T. Eiter, G. Ianni, T. Krennwallner, Answer set programming: A primer, in: S. Tessaris, E. Franconi, T. Eiter, C. Gutiérrez, S. Handschuh, M. Rousset, R. A. Schmidt (Eds.), Reasoning Web. Semantic Technologies for Information Systems, volume 5689 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 40–110. URL: https://doi.org/10.1007/978-3-642-03754-2_2. doi:10.1007/978-3-642-03754-2_2.

[10] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot asp solving with clingo, Theory and Practice of Logic Programming 19 (2019) 27–82. doi:10.1017/S1471068418000054.

[11] R. David, S. Bischof, K. Diwold, J. X. Parreira, Symbolic-ai driven data repairs for large scale energy co-simulations (2025).