Semantic Web 1 (0) 1–5
IOS Press

RDF Graph Validation Using Rule-Based Reasoning

Ben De Meester [a,*], Pieter Heyvaert [a], Dörthe Arndt [a], Anastasia Dimou [a], and Ruben Verborgh [a]

[a] *Ghent University – imec – IDLab, Department of Electronics and Information Systems, Technologiepark-Zwijnaarde 19, 9052 Ghent, Belgium*
E-mails: ben.demeester@ugent.be, pheyvaer.heyvaert@ugent.be, doerthe.arndt@ugent.be, anastasia.dimou@ugent.be, ruben.verborgh@ugent.be

Abstract. Semantic Web applications cannot function when the given data – i.e., an RDF graph – is not interpreted as expected. RDF graphs can be validated by defining and assessing constraints. These constraints define an RDF graph that can be correctly interpreted for a specific Semantic Web application or use case. Which entailment regime is used – e.g., whether rdfs:subClassOf inferencing is taken into account or not – is an integral part of how the RDF graph should be interpreted, and thus of the proper functioning of the application. Different types of validation approaches are proposed to assess these constraints, namely hard-coded systems, ontology reasoners, and querying endpoints. However, these approaches do not allow to fully customize which inferencing is supported to match the entailment regimes as intended by the use case. They are thus unable to validate RDF graphs properly or need to combine systems, deteriorating the performance of the validation. In this paper, we present an alternative validation approach using rule-based reasoning, capable of fully customizing the inferencing rules during validation. We compare existing approaches with a rule-based reasoning approach, and present both a formal ground and practical implementation based on N3Logic and the EYE reasoner. Our approach (a) better explains the root cause of the violations due to the formal logical proof of the reasoner, (b) returns an accurate number of violations due to explicit inferencing rules, and (c) supports more constraint types by including inferencing up to at least OWL-RL complexity and expressiveness. Moreover, our performance evaluation shows that our implementation is faster than combining existing approaches. By allowing to precisely define the inferencing rules together with the constraints, we provide a more complete validation approach. We ensure validated RDF graphs can be interpreted as intended with additional inferencing, allowing more precise Semantic Web applications, and opening opportunities for automatic RDF graph refinement and validating implicit graphs based on their generation rules.

Keywords: Constraints, Rule-based Reasoning, Validation

1. Introduction

Semantic Web data is represented using the Resource Description Framework (RDF) [1], forming an *RDF graph* [1]. Semantic Web applications however cannot function if given RDF graphs' intended meaning are not interpreted as expected by the use case, i.e., the graph does not adhere to a certain schema or shape. This relates to their *quality*: RDF graphs which are interpreted as expected are more "fit for use" and thus of higher quality [2]. RDF graphs can be (automatically) *validated* by assessing *constraints* [3]. These constraints and their intended meaning depend on the use case and are stated explicitly. Via validation, we can thus discover invalid (portions of) RDF graphs: the *violations* that are returned. These violations guide the user to discover and refine the violating resources and relationships (i.e., the *root causes*), resulting in an RDF graph of higher quality [4]. This makes sure that RDF graphs can be properly interpreted for a specific Semantic Web application or use case [5].

1.1. Problems

Let us consider the following example: an RDF graph containing people and their birthdates is validated. Due to the use case, it is important that "the RDF graph

*Corresponding author. E-mail: ben.demeester@ugent.be.

only contains persons, their name, and their birthdate". Other or incomplete data should result in a violation. Specifically, we validate an RDF graph such as shown in formula (1)[1], with a relevant ontology represented in formula (2).

```
:Bob a :Man ;
  :firstname "Bob" ;
  :birthdate "01-01-1970"^^xsd:date .   (1)
:birthdate rdfs:domain :Person .        (2)
:Bob a :Person .                        (3)
```

Problem 1 (P1): it is hard to find the root causes of the constraint violations. For example, a use case dictates that a violation should be returned when the following compound constraint is not met: given a resource r, this resource has $(r_{firstname} \wedge r_{lastname}) \vee (r_{nickname})$. When a violation is returned, all resources that do not conform to this constraint are marked. However, it is not always clear which part of the constraint caused the violation: does the resource lack firstname, lastname, or nickname? In the case of formula (1), either a lastname or nickname is missing, however, current approaches only report *which* resources violate which constraints, not *why* the violation occurs. If it is hard to understand why a constraint was violated (i.e., to find the root cause), it is hard to refine the RDF graph and improve its quality.

Problem 2 (P2): the number of found violations is biased with respect to the used entailment. The used entailment regime [1] – e.g., taking rdfs:subClassOf inferencing [6] into account or not – is an integral part of how the RDF graph should be interpreted, but also validated. A mismatch between the entailment regime supported during validation and the entailment regime as intended by the use case influences, in this case, whether formula (3) is inferred or not. Thus, either too many or too few violations can be returned [7]. This biased number of violations gives a biased idea of the real quality of the validated RDF graph.

Many violations: formula (2) specifies the domain of `:birthdate`. We validate the constraint of "the RDF graph only contains persons, and contains a birthdate for every person" given formula (1). When the used entailment regime does not support formula (2), this would result in a violation that formula (3) is missing in the RDF graph. However, when the used entailment regime does support formula (2), we can infer formula (3). The violation is *resolved early-on*, and no violation is returned.

Few violations: Assume that the RDF graph of formula (1) also contains formula (4) and formula (3) is not explicitly stated.

```
:Bob a :Dog .                           (4)
:Dog owl:disjointWith :Person .         (5)
```

In a system that does not support inferring statements due to formula (2) and formula (5), no additional violation would be found. However, supporting those entailments can create new statements to be validated, and *find implicit violations*. By inferring formula (3) using formula (2) and supporting formula (5), an inconsistency is returned because `:Bob` cannot be a `:Dog` and `:Person` at the same time.

Problem 3 (P3): not all constraint types are supported. Not all existing validation approaches support all constraint types. Approaches that assume Closed World Assumption (CWA) support different constraint types than approaches that assume Open World Assumption (OWA). For example, approaches supporting OWA cannot be used to find out whether a specific triple is *not* present in an RDF graph. Inferencing support varies largely across approaches, influencing validation [7]. Current approaches either only support a fixed set of inferencing rules, or no inferencing at all, let alone complex cases involving inferencing which are not (well) supported. Hartmann et. al has shown that thirty-five out of the eighty-one constraint types (43.2%) are constraint types that benefit from including inferencing.

Different types of validation approaches are proposed, namely using *hard-coded systems*, *ontology reasoners*, and *querying endpoints*. Existing approaches inhibit the aforementioned problems. They are not able to customize the used inferencing rules for validation, thus the employed semantics are usually implicit. Due to this bias, the reported quality could be higher or lower, depending on the employed semantics of the validation approach (~P2). *Hard-coded systems* couple the business logic within the code base, i.e., a black box: they cannot fully indicate the root cause without inspecting the code base (P1). *Ontology reasoners* can only partially customize the used entailment regime, and do no support all constraint types [7] (~P2, ~P3).

[1] For the remainder of the paper, empty prefixes denote the fictional schema http:example.com/, other prefixes conform with the results of https://prefix.cc.

Querying endpoints are also a black box, can only partially customize the used entailment regime, and do no support all constraint types [7] (P1, ~P2, ~P3).

Problem 4 (P4): Combining inferencing deteriorates performance P3 could be solved by performing an inferencing step as a preprocessing step prior to validation [7], thus combining multiple systems. However, (i) the combination is still a black box: it is not clear whether a piece of RDF data came from the original dataset or was inferred (P1); (ii) the preprocessing step possibly generates data not relevant for validation [7], or generates data which semantics conflict with the validation semantics (P2); and (iii) the combination slows down the overall system (P4).

1.2. Hypotheses

The aforementioned observed problems lead to the following hypotheses:

Hypothesis 1: a validation approach using a declarative logic explains constraint violations' root causes accurately in more cases than query-based or hard-coded validation approaches.

Hypothesis 2: a validation approach with customizable inferencing returns an accurate number of validation results with respect to the used entailment regime.

Hypothesis 3: a declarative validation approach with custom inferencing supports more constraint types than existing approaches.

Hypothesis 4: a declarative validation approach that supports custom inferencing is faster than a validation system that includes the same inferencing as a preprocessing step.

1.3. Contributions

In this paper, we propose a rule-based reasoner approach for RDF graph validation. Rule-based reasoners can generate a proof stating which rules were triggered for which returned violation. Thus, the root causes of violations can be clearly explained (solving P1). As opposed to ontology reasoners, they natively support the inclusion of *custom* inferencing rules, which means the supported inferencing can be explicitly stated. For example, when validating a specific use case, disjointness is ignored, domain and range is supported, and other valid inferencing rules are taken into account. Validation then returns an accurate number of found violations and supports more constraint types (solving P2 and P3). Moreover, rule-based reasoners only need a single language to declare both the constraints and inferencing rules, and only a single system to execute the validation. Thus, this approach is faster than including an inferencing preprocessing step (solving P4).

Our contributions are as follows.

i We analyze existing validating approaches, and compare to a rule-based reasoning approach.
ii We provide a formal ground for using rule-based reasoning for validation.
iii We provide an implementation using N3Logic and the EYE reasoner.
iv We validate our hypotheses functionally and evaluate performance, positioning our work within the state of the art.

We validated that (a) the formal logical proof explains the root cause of a violation more detailed than the state of the art; (b) an accurate number of violations is returned due to explicit inferencing rules; (c) customizable inferencing rules allow supporting more constraint types up to at least OWL-RL complexity and expressiveness; and (d) our performance evaluation shows that our implementation is faster than a combined system, and faster than an existing validation approach when RDF graphs are smaller than one hundred thousand triples.

The remainder of the paper is organized as follows: We provide an overview of the state of the art in Section 2, after which we position rule-based reasoning as an alternative and compare in Section 3. We provide a formal ground (Section 4) and practical implementation (Section 5), and evaluate in Section 6. We conclude in Section 7.

2. State of the art

Since this work encompasses both validation and reasoning, and proposes an alternative validation approach, we first provide a background on validation and reasoning in Section 2.1. Then, we give an overview of existing validation approaches in Section 2.2, and of related vocabularies and ontologies in Section 2.3. Current approaches provide overlapping functionality, thus, we conclude with an overview of general constraint types in Section 2.4. Our categorization is derived from the general quality surveys of Zaveri et al. [8], Ellefi et al. [9], and Tomaszuk [10]. The related works from those surveys are extended with recent

works published in, a.o., the major Semantic Web conferences (ESWC and ISWC), and the major Semantic Web journals (Journal of Web Semantics and Semantic Web Journal).

2.1. Background

Validation Quality is commonly defined as "fitness for use" [2]. Data quality can be assessed by employing a set of *data quality assessment metrics* [11]. Quality assessment for the Semantic Web – and more specifically, for Linked Data – spans multiple dimensions, further categorized in *accessibility*, *intrinsic*, *trust*, *dataset dynamicity*, *contextual*, and *representational* dimensions [8]. Validating an RDF graph directly relates to intrinsic quality dimensions, as defined by Zaveri et al.: (i) independent of the user's context, and (ii) checking if information correctly and compactly represents the real world data and is logically consistent in itself [8], i.e., the graph's adherence to a certain schema or shape [8]. In this paper, we specifically focus on RDF graph validation, i.e., the intrinsic dimensions.

Validation of an RDF graph can be automated by checking a set of *constraints* using a set of *test cases* [3], each assessing a specific *constraint*. *Constraint violations* are then indicated when a validation returns negative results. For instance, a validation assesses for a specific RDF graph if all objects linked via the predicate `schema:birthdate` are a valid `xsd:date`, or if all subjects and objects linked via the predicate `foaf:knows` are explicitly listed to be of type `ex:Human`. Negative results are returned, indicating violations. It should be clearly noted that there is a distinction between constraints and what is described in an ontology. For example, the FoaF ontology[2] declares the domain and range of the `foaf:knows` predicate as `foaf:Person`. When taking `rdfs:domain` and `rdfs:range` entailment into account, any resource linked via `foaf:knows` is inferred as a `foaf:Person`. Whether those triples are then mentioned explicitly or not does not contribute to a violation when only described in an ontology.

Reasoning A *reasoner* is a piece of software that performs *reasoning*: inferring logical consequences (an *inference*) from a set of asserted facts [12]. Asserted facts (*axioms*) are commonly annotated using an *ontology language*. Examples of ontology languages are RDF Schema (RDFS) [6] and the Web Ontology Language (OWL) [13]. The *inferencing rules* defining the reasoning are then specified, as this ontology language follows a certain *logic*. A fixed set of inferencing rules that specify a specific (description) logic is called an *entailment regime* [14]. Reasoning on Web ontology languages is commonly done using *description logics*. In the case of the Semantic Web, OWL-DL with subprofiles like OWL-RL and OWL-QL prevail [13]. An ontology language can then be used to describe, e.g., that "Every Man is a Person", and the employed description logic defines which consequences should be inferred based on that description.

Semantic Web reasoners can be *ontology reasoners* or *rule-based reasoners*. Ontology reasoners are optimized for specific description logics, such as KAON2[3] and FaCT++[4]. Rule-based reasoners typically follow two types of inferencing algorithms: *forward chaining* and *backward chaining* [15]. Whereas forward chaining tries to infer as much new information possible, backward chaining is goal-driven: the reasoner starts with a list of goals and tries to verify whether there is background knowledge and rules available that support any of these goals [15]. The employed rules are the logic the reasoner, such as EYE [16] or cwm [17], follows. Whereas ontology reasoners have the inferencing rules for, e.g., `rdfs:subClassOf` and other RDFS or OWL constructs embedded, rule-based reasoners commonly rely on the general "implies" construct. Each rule thus specifies "A implies B", where both the *antecedent* "A" and the *consequence* "B" can consist of statements [15]. Existing constructs such as `rdfs:subClassOf` can be translated into one or more rules[5].

2.2. Validation Approaches

In this section, we discuss RDF graph validation approaches: the constraint languages and the approaches used to implement them (summarized in Table 1). Other tools and surveys that cover other quality dimensions such as accessibility or representational dimensions are out of scope, given that we focus on validation. First came *hard-coded* systems. Then, *constraint languages* were proposed, as a declarative means to describe RDF graph constraints. RDF graph validation approaches that support constraint languages can be categorized as *integrity constraints*, *query-based constraint languages*, or *grammar-based constraint languages*.

[2] http://xmlns.com/foaf/spec/

[3] http://kaon2.semanticweb.org/

[4] http://owl.cs.manchester.ac.uk/tools/fact/

[5] http://eulersharp.sourceforge.net/#theories

Table 1
Which validation approach implements which language

	hard-coded system	ontology reasoner	querying endpoint
(no language)	✓		
integrity constraints		✓	✓
query-based			✓
grammar-based	✓		✓

2.2.1. Hard-coded

In hard-coded systems, the implementation embeds both description and validation of constraints. Hogan et al. analyzed common quality problems both for publishing and intrinsic quality dimensions [18], providing an initial set of best practices [19]. Efforts focus on a limited set of configurable settings (turning constraint rules on or off), dependent on the implementation [20].

2.2.2. Integrity Constraints

Integrity constraints define the adherence of an RDF graph to a certain schema, in terms of the used vocabularies and ontologies [21]. For RDF graph validation, the used ontologies are *interpreted as integrity constraints* [21–23]. Integrity constraint validation approaches either use an ontology reasoner or a querying endpoint.

Ontology reasoner Motik et al. [22] propose semantic redefinitions, where a certain subset of TBox axioms are designated as constraints. They thus propose alternative semantics for OWL. To know which semantics apply, constraints have to be marked as such [22]. They propose to integrate their implementation with KAON2. Furthermore, custom integrity constraints for Wordnet have been verified using Protégé [24] with FaCT++ [25].

Querying endpoint Tao et al. [21] propose using OWL expressions with Closed World assumption and a weak variant of Unique Name assumption to express integrity constraints. OWL semantics are redefined, without being explicitly stated as such during validation. They use the SPARQL query language [14] for RDF, RDFS, and OWL-DL entailment regimes [21], e.g., using SPARQL property paths to simulate `rdfs:subClassOf` entailment. Tao et al. work in a general OWL setting, where their approach is sound but not complete. In an RDF setting the approach is both sound and complete, as there is only a single model that needs to be considered [23]. This implementation is incorporated into Stardog ICV [26]. Schneider separates validation into integrity constraints and Closed World recognition [23], showing that RDF and RDFS entailment can be implemented for both by translation to SPARQL queries.

2.2.3. Query-based

Query-based languages define the RDF graph's structure. Constraints are thus interpreted similar to SPARQL queries [27]: only RDF graphs whose structure is compatible with the defined structure are returned. These approaches rely on an embedded or external querying endpoint, and either require explicitly writing SPARQL(-like) queries to define the constraints, or make use of a higher-level language.

Query-based validation approaches that require explicitly writing SPARQL(-like) queries to define the constraints include CLAMS [28], SPIN [29], and the work of Kontokostas et al. [3]. CLAMS [28] is a system to discover and resolve inconsistencies in Linked Data. They define a violation as a minimal set of triples that cannot coexist. The system identifies all violations by executing a SPARQL query set. Knublauch et al. [29] propose the SPARQL Inference Notation (SPIN): a SPARQL-based rule and constraint language. The SPARQL query is described using RDF statements instead of using the original SPARQL syntax. Kontokostas et al. [3] propose Data Quality Test Patterns (DQTP): tuples of typed pattern variables and a SPARQL query template to declare test case patterns. The validation framework that validates these DQTPs is called RDFUnit [3]. The DQTPs are transformed into SPARQL queries, where every SPARQL query is a test case. RDFUnit additionally allows automatically generated test cases, depending on the used schema.

RDFUnit is also used to validate Linked Data generation rules in the RDF Mapping Language (RML) [4], by manually defining different DQTPs to target also generation rules instead of only the generated RDF graph. This means the RDF graph can be validated before data is even generated, as these rules reflect how the RDF graph will be formed when generated.

SPARQL can thus be used to describe validation. however, as SPARQL is not targeted towards validation, there is need for a terse, higher-level language [30]. Related work includes Luzzu [31] and SHACL [32]. Luzzu [31] uses a custom declarative constraint language (Luzzu Quality Metric Language, LQML). Any metric that can be expressed in a SPARQL query can be defined using LQML [31]. Moreover, other quality dimensions except the intrinsic dimensions are also expressible using LQML [31]. Luzzu supports basic metrics and custom JAVA code allowing users to im-

plement custom metrics. The Shapes Constraint Language (SHACL) became a W3C Recommendation for validating RDF graphs against a set of constraints (the so-called *data shape*) [32]. The core of SHACL is independent of the SPARQL syntax, which promotes the development of new algorithms and approaches to validate RDF graphs [30]. Querying SHACL implementations include Corese SHACL[6] – a SPARQL endpoint that can resolve SHACL constraints on top of a loaded RDF graph – and the TopBraid SHACL API[7]. These implementations roughly follow the same approach. First, an external querying endpoint is used, or a querying endpoint – using, e.g., Apache Jena[8] – is set up. This querying endpoint accommodates the test cases that are executed as queries. Second, in the case of the embedded querying endpoint, the data and background knowledge – comprising ontologies, vocabularies, and additional RDF graphs – is loaded in the endpoint, forming the knowledge base. Loading data in a querying endpoint is usually accompanied by filling additional indexes to improve querying time [33]. Third, a test set is retrieved, declared in SHACL. Fourth, the validation takes place.

2.2.4. Grammar-based

Grammar-based approaches define a domain specific language to declare validation rules [30]. Grammar-based validation approaches either use a querying endpoint or are hard-coded.

Description Set Profiles (DSP) [34] define a set of constraints using Description Templates, targeted specifically to Dublin Core Application Profiles. SPIN has been used to implement DSP [35]. OSLC Resource Shapes [36] – which became part of IBM Resource Shapes – are proposed as a high level and declarative description of the expected contents of an RDF graph expressing constraints. Fischer et al. [37] proposed RDF Data Descriptions as another domain specific language that is compiled to SPARQL.

Shape Expressions (ShEx) [38] is another grammar-based approach which defines a domain specific language for RDF graph validation. The grammar of ShEx is inspired by Turtle and RelaxNG, and its complexity and expressiveness are formalized [39, 40]. ShEx does not rely on an underlying technology such as SPARQL to perform the validation, a hard-coded system is used instead [30].

2.3. Validation reports

General quality assessment tools give an indication of the overall quality across all data quality dimensions. This is typically returned in the form of a report. An initial step was the Dataset Quality Ontology (daQ) [41], which is extended by Radulvic et al. [42] to include all data quality dimensions as identified by Zaveri et al. [8], and led to the Data Quality Vocabulary [41]. These efforts describe the relevant data quality dimensions, so that different frameworks can be compared.

On the other hand, the constraint violations report itself makes use of (other) specific vocabularies. This report thus usually refers to dimensions as specified using the aforementioned vocabularies. Violations returned by validation tools guide the user to discover and refine the violating resources and relationships.

Kontokostas et al [3] used the RDF Logging Ontology[9] (RLOG) to describe the results of the violations of RDFUnit. RLOG is derived from the Log4j framework[10] to describe test case results. More specific reporting ontologies include the Reasoning Violations Ontology (RVO) for integrity constraint violations [43], the Quality Problem Report Ontology for assembling detailed quality reports for all data quality dimensions [31], and the SHACL report format [32]. These formats report on the intrinsic data quality dimensions, and allow to distribute and compare the violations found in an RDF graph.

2.4. Constraint types

Hartmann et al. identified eighty-one general *constraint types* [44]. These constraint types are an abstraction of specific constraints, independent of the constraint language used to describe them. On the one hand, a constraint type can be defined in different ways. For example, the *property domain* constraint type specifies that resources that use a specific property should be classified via a specific class, e.g., all resources using the `:birthdate` property that are not classified as a `Person` are violating resources. Using RDFS [6], the property domain constraint type can be assessed by interpreting `rdfs:domain` as an integrity constraint. Using SHACL, this can be achieved by defining a `sh:property` with `sh:class` for a `sh:targetSubjectsOf` shape [32]. On the other hand, the analysis of Hartmann et al. unveiled that certain

[6] http://wimmics.inria.fr/corese
[7] https://github.com/TopQuadrant/shacl
[8] https://jena.apache.org/
[9] http://persistence.uni-leipzig.org/nlp2rdf/ontologies/rlog#
[10] https://logging.apache.org/log4j/

constraint types are only describable by certain constraint languages [44], e.g., SHACL currently does not support all constraint types [45].

Moreover, Hartmann et al. provide a logical underpinning stating the requirements for a validation approach to support all constraint types [7]. For thirty-five out of eighty-one constraints, reasoning (up to OWL-DL expressiveness) can improve the validation: violations can be resolved early-on and implicit violations can be found. Supporting inferencing rules is thus an important requirement for validation approaches.

3. Comparative analysis

In this section, we show the shortcomings of existing approaches, and explain how rule-based reasoning as a validation approach can address them. Our analysis is summarized in Table 2.

We adapt the framework as presented by Pauwels et al. [46], which introduced comparative factors of key implementation strategies for compliance checking applications. These factors provide for a comparative framework across validation approaches. We generalize the factors *inferencing rules*, *time*, and *customization*. We further introduce *explanation* and *reasoning preprocessing* as validation-specific factors.

Inferencing rules Inherent support for (custom) inferencing rules [46]. *Hard-coded systems* might support fixed inferencing rules, but this cannot be inspected or altered without investigating the code base. *Ontology reasoners* support specific entailment regimes. However, using them for validation is limited to inferencing failures, which is a very limited form of validation. For example, disjointness will force ontology reasoners to throw an error, which can then be interpreted as a violation. Approaches that do use OWL as a constraint language change the semantics of OWL to include – among others – some form of CWA. This leads to ambiguity in the Semantic Web as an existing, globally agreed upon logic is changed [47], and thus makes it no longer possible to combine validation with OWL reasoning because the meaning is different. *Querying endpoints* support up to RDF and RDFS entailment via translation of the SPARQL queries using property paths [21, 23]. Using SPARQL property paths thus provides only limited inferencing expressiveness. Also, performance deteriorates [48]. Rule-based reasoners allow additional (user-defined) rules [15]. By including the relevant rule set, different description logics and (user-defined) logics can be supported. Rule-based reasoners thus allow to fully customize the inferencing rules.

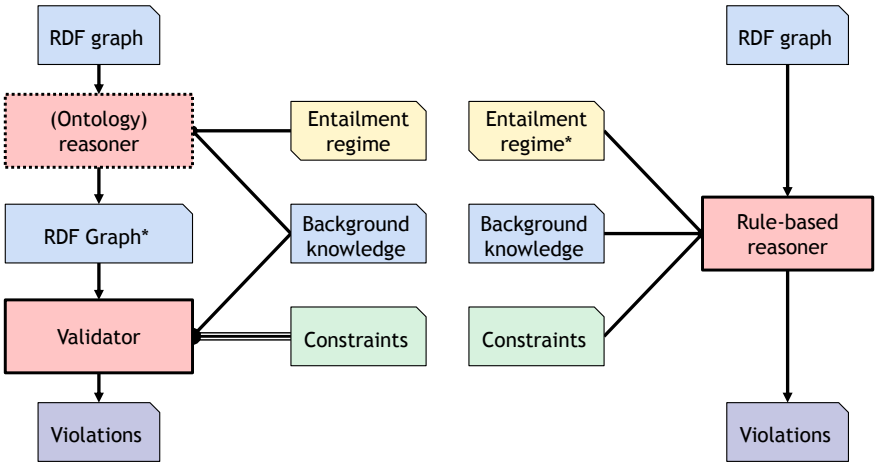
Explanation How specific a constraint violation can be explained. *Hard-coded systems* can only provide explanations if it is included in the code base (i.e., a *black box*). The *querying endpoint* also acts as a black box. Explanations coming from these black boxes cannot be verified without inspection the code base. Also for *ontology reasoners*, it is not a standard feature to produce proofs of their results [49]. *Rule-based reasoners* however can commonly provide a logical proof, as they do not contain specific description logic optimizations and rely on a general "implies" construct to describe rules. The logical proof then declares which rules were triggered to come to a certain conclusion. This proof can give a precise explanation for the root cause of constraint violations. Whereas other approaches rely on changes in the code base to provide a proof, rule-based reasoners provide a logical framework that inherently supports this proof.

Time The execution time: short versus long [46]. *Hard-coded systems* are usually faster and need shorter processing time [46], as they can be optimized for specific use cases or constraint languages. *Ontology reasoners*, *querying endpoints*, and *rule-based reasoners* typically remain independent of the constraint language, but are also slower than hard-coded systems [46]. However, except for ontology and rule-based reasoners, the other approaches require a reasoning preprocessing step to combine inferencing with validation, which influences the total validation time (as indicated by an asterisk in Table 2).

Customization The extent of customization each type of approach enables [46]. Customization of *hard-coded systems* is limited without requiring a development effort [50], as the business logic is embedded within the code. The other approaches allow for declarations to customize the validation. Declarations are decoupled, i.e., independent of the tool's implementation. Thus, they can be shared and easier customized to a certain use case. *Ontology reasoners* support specific entailment regimes. One or more reasoning profiles such as OWL-QL and OWL-DL are thus built into the ontology reasoner. *Query endpoints* allow customization by defining additional SPARQL queries, as long as no additional reasoning is needed [46]. For *rule-based reasoners*, a single system suffices to support explicit semantics, as inferencing rules can be added and removed depending on the use case [46].

Table 2
Summarizing the different validation approaches, and comparing them with rule-based reasoning

Approaches	Hard-coded system	Ontology reasoner	Querying endpoint	Rule-based reasoner
Inferencing rules	No / Limited	Yes*	No / Limited	Yes
Explanation	No	Yes	No	Yes
Time	Short*	Long	Long*	Long
Customization	Limited	Limited	Open	Open
Reasoning preprocessing	Yes	Limited	Yes	/



(1) Preprocessing approach (2) Rule-based reasoning approach

Figure 1. The preprocessing approach: first (optionally, hence the dashed line), an *(Ontology) reasoner* is used to generate intermediate data (*Data**). That intermediate data is then the input data for the *Validator*. Using a rule-based reasoner only needs a single system and language to combine reasoning and validation.

Reasoning preprocessing Supporting a specific entailment regime for *hard-coded systems* and *querying endpoints* can be enabled by including a reasoning step as preprocessing step (see Fig. 1.1) [7]: An (ontology) reasoner first reasons over the original RDF graph (Fig. 1.1, *(Ontology) reasoner*, optionally, hence the dashed line), after which the newly generated RDF graph is used for validation (Fig. 1.1, *Validator*). This could also prove beneficial for *ontology reasoners*, allowing to combine the original semantics during the preprocessing with the altered semantics for interpreting the integrity constraints.

However, using a preprocessed inferred RDF graph has following disadvantages: (i) multiple systems need to be combined and maintained, e.g., a reasoner and a querying endpoint; (ii) different languages need to be learned and combined for the inferencing rules and constraints (e.g., OWL and SPARQL); (iii) as these multiple systems are not aligned, the reasoner could generate a large number of new triples that are irrelevant to the defined constraints, which could lead to bad scaling (Fig. 1.1, *RDF graph**); and (iv) the explanation is hindered even more: there is no distinction whether a violation is due to data inferred from the reasoning step, or due to the original RDF graph. Querying endpoints that have embedded support for entailment regimes would not inhibit disadvantages (i) and (iii), however, they still rely on the combination of multiple languages and hinder the root cause specification.

Using a *rule-based reasoner* does not require reasoning preprocessing. Both the inferencing rules and constraints can be defined using the same declaration using inferencing rules (Fig. 1.2), and executed simultaneously on the RDF graph and the background knowledge (i.e., the referenced ontologies, vocabularies, and additional data). Moreover, a rule-based reasoner natively supports custom inferencing rules, and thus, custom entailment regimes (Fig. 1.2, *Entailment regime**).

4. Logical Requirements

In this section, we discuss which requirements a logic needs to be a good choice for RDF graph validation, and argue for using a rule-based logic. Constraint languages need to cope with very different constraint types depending on users' needs. Each constraint type implies certain logical requirements. The constraint types and requirements they entail are investigated by Hartmann et al., who claimed that the Closed World Assumption (CWA) and Unique Name Assumption (UNA) are crucial for validation [7]. These requirements are not common for Semantic Web logics, as data on the Web is decentralized and information is spread ("anyone can say anything about anything" [1]), and single resources can have multiple URIs. Hartmann et al. emphasize the difference between reasoning and validation, and favor SPARQL-based approaches for validation. The latter – when needed – can be combined with, e.g., OWL-DL or OWL-QL reasoning as a preprocessing step.

However, we show that Semantic Web rule-based reasoners *can* be used for validation, even though they – due to the reasons mentioned above – normally do not follow CWA and UNA. Specifically, we state that the requirements for using a rule-based reasoner are (i) supporting Scoped Negation as Failure (SNAF) instead of CWA (Section 4.1), (ii) containing predicates to compare URIs and literals instead of supporting UNA (Section 4.2), and (iii) supporting expressive built-ins, as validation often deals with, e.g., string comparison and mathematical calculations (Section 4.3).

4.1. Scoped Negation as Failure

Existing works claim that CWA is needed to perform validation [7, 21, 23]. Given that most Web logics assume OWA, this would require semantic redefinitions to include inferencing during validation [22], which leads to ambiguity. However, as validation copes with the *local knowledge base*, and not the entire Web, we claim Scoped Negation as Failure (SNAF) [51–53] is sufficient. This is an interpretation of logical negation: instead of stating that ρ does not hold (i.e., $\neg\rho$), it is stated that reasoning fails to infer ρ within a specific *scope* [51–53]. This scope needs to be explicitly stated.

To understand the idea behind Scoped Negation as Failure, let us validate following RDF graph:

```
:Kurt a :Researcher;                    (6)
      :name "Kurt01".                   (7)
```

We validate the constraint "every individual which is declared as a researcher is also declared as a person". This thus means a violation is returned when an individual is found during validation which is a researcher, but not a person:

$$\begin{aligned}\forall \texttt{x} : &(\texttt{(x a :Researcher)} \wedge \\ &\neg\texttt{(x a :Person)}) \\ \rightarrow &\texttt{(:constraint :isViolated "true".)}\end{aligned} \tag{8}$$

As stated, this constraint cannot be tested with OWA: the knowledge base contains the triple of formula (6), but not of:

```
:Kurt a :Person.                        (9)
```

The rule is more general: given its open nature, we cannot guarantee that there is no document in the entire Web which declares the triple of formula (9).

This changes if we take into account SNAF. Suppose that $\mathcal{K}$ is the set of triples we can derive (either with or without reasoning) from our knowledge base of formulas (6) and (7). Having $\mathcal{K}$ at our disposal, we can test:

$$\begin{aligned}\forall \texttt{x} : &((\texttt{(x a :Researcher)} \in \mathcal{K}) \wedge \\ &\neg(\texttt{(x a :Person)} \in \mathcal{K})) \\ \rightarrow &\texttt{(:constraint :is :violated.)}\end{aligned} \tag{10}$$

The second conjunct is not a simple negation, it is a negation with a certain scope, in this case $\mathcal{K}$. If we add new data to our knowledge base, e.g., the triple of formula (9), we would have a different knowledge base $\mathcal{K}'$ for which other statements hold. The truth value of formula (10) would not change since this formula explicitly mentions $\mathcal{K}$. SNAF is what we actually need for validation: we do not validate the Web in general, we validate a specific RDF graph.

4.2. Predicates for Name Comparison

UNA is deemed required for validation [7], i.e., every resource taken into account can only have one single name (a single URI in our case) [54]. UNA is in general difficult to obtain for the Semantic Web and Web logics due to its distributed nature: different RDF graphs can – and actually do – use different names for

the same individual or concept. For instance, the URI `dbpedia:London` refers to the same place in Britain as, e.g., `dbpedia-nl:London`. That fact is even stated in the corresponding datasets using the predicate `owl:sameAs`. The usage of `owl:sameAs` conflicts with UNA and influences validation [7].

Let us look into the following example. We assume `dbo:capital` is an `owl:InverseFunctionalProperty`. Our knowledge base contains:

```
:Britain dbo:capital :London.          (11)
:England dbo:capital :London.          (12)
```

Since both `:Britain` and `:England` have `:London` as their capital and `dbo:capital` is an inverse functional property, an ontology reasoner would derive that

```
:Britain owl:sameAs :England.          (13)
```

This thus influences the validation result. Such a derivation cannot be made if UNA is valid, since UNA explicitly excludes this possibility.

The related constraint – defined as `INVFUNC` by Kontokostas et al. [3] – specifies that each resource should contain exactly one relationship via `dbo:capital`, i.e., the capital is different for every resource. The constraint `INVFUNC` is related to `owl:InverseFunctionalProperty`, but it is slightly different: while OWL's inverse functional property refers to *the resources* that are in the domain of `dbo:capital`, the validation constraint `INVFUNC` refers to *the representation of those resources*. The RDF graph of formulas (11) and (12) thus violates the `INVFUNC` constraint. Even if our logic does not follow UNA, this violation can be detected if the logic offers predicates to compare the (string) representation of resources.

4.3. Expressive Built-ins

Validation often deals with, e.g., string comparison and mathematic calculations. These functionalities are widely spread in rule-based logics using *built-in functions*. While it normally depends on the designers of a logic which features are supported, there are also common standards. One of them is RIF, whose aim is to provide a formalism to exchange rules in the Web [55]. Being the result of a W3C working group consisting of developers and users of different rule based languages, RIF can also be understood as a reference for the most common features rule based logics might have.

Let us take a closer look to the comparison of URIs from the previous section. `func:compare` can be used to compare two strings. This function takes two string values as input, and returns `-1` if the first string is smaller than the second one regarding a string order, 0 if the two strings are the same, and 1 if the second is smaller than the first. The example above gives:

```
("http://example.com/Britain"
 "http://example.com/England")
                    func:compare -1.   (14)
```

To refer to a URI value, RIF provides the predicate `pred:iri-string` which converts a URI to a string and vice versa. To enable a rule to detect whether the two URI names are equal or not, an additional function is needed: the reasoner has to detect whether the comparison's result is different from zero. That can be checked using the predicate `pred:numeric-not-equal`, which is the RIF version of $\neq$ for numerical values. In the example, the comparison would be `true` since $0 \neq -1$. Using these RIF built-ins, a reasoner can check the name equality between `:Britain` and `:England`, and return a violation. Whether a rule based Web logic is suited for validation highly depends on its built-ins. If it supports all RIF predicates, this can be seen as a strong indication that it is expressive enough.

5. Application

In this section, we explain how we applied a rule-based reasoner for validation. We discuss the different configurable components and the workflow that make up a fully customizable validation in Section 5.1. Then, we discuss the technologies involved in Section 5.2 and implementation in Section 5.3. We end with an example using rules at Section 5.4.

5.1. Customizable validation

A use-case specific validator is made up of multiple configurable components(Fig. 2). The used *entailment regime* explicitly specifies the graph's interpretation during the validation. This is a set of rules that form either existing entailment regimes such as RDFS [6], or a custom entailment regime. The set of rules that form the *constraint translation* infers the general constraint types from specific constraint descriptions.

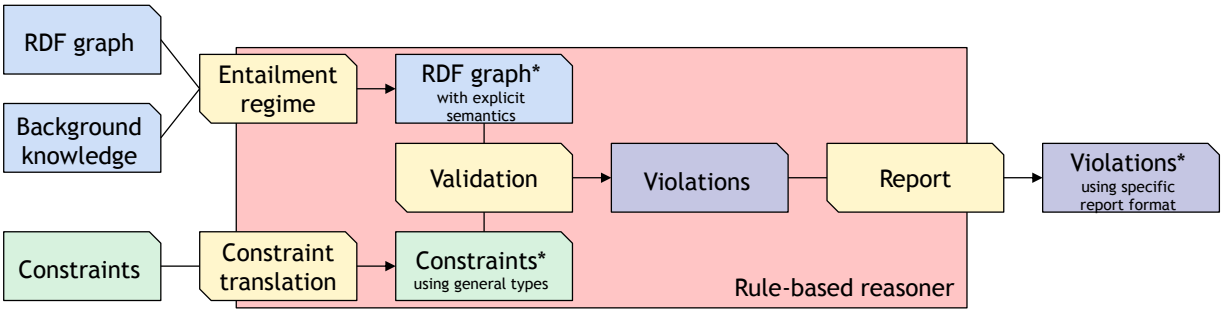


Figure 2. Components view. All double-snipped rectangles are inferencing rule sets, the remaining are data or constraints. Four parts can be identified within the validation execution: (i) the RDF graph – together with optional background knowledge – is interpreted correctly using a (custom) entailment regime, (ii) the general constraint types are inferred from the given constraints, (iii) the actual validation takes place, and (iv) the returned violations are structured given a (custom) report format.

There are general constraint types, common across existing constraint languages [44]. We can infer these types from the constraints described in a specific language such as SHACL [32], or a custom language. We use RDF-CV[11] [5] to describe the general constraint types. RDF-CV generalizes the constraint types into a coherent structure. The set of rules that form the *report format* infers the resulting violations in the format specified by an existing report vocabulary, e.g., the SHACL report format [32], or a custom report vocabulary.

This declarative approach is decoupled from ontologies, constraint languages, and report formats, and customizable to different use cases. The resulting validator can thus be fully customized, or not customized at all by not including any aforementioned rule sets. Not including any rule sets results in a validator that does not employ any additional inferences, only validates constraints described using general constraint types, and returns a report in a general format, based on the same structure as RDF-CV.

The validator's input comprises sets of RDF statements (Fig. 2, left): the *RDF graph* to be validated, the *background knowledge* – comprising ontologies, vocabularies, and any other additional RDF graphs –, and validation constraints. The output is again a set of RDF statements, containing the violations (Fig. 2, right).

A single inferencing execution will take into account all previous rule sets and sets of RDF statements. The correct interpretation is inferred using the entailment regime, taking into account the RDF graph and background knowledge. The general constraint types are inferred from the constraints. The validation itself uses a fixed set of rules, taking into account the explicitly defined interpretation of the RDF graph and the general constraint types. The output consists of the general violations transformed to a specific report format.

5.2. Technologies

The most important technological considerations are the rule-based web logic and reasoner in accordance with that logic. Rule-based web logics include the Semantic Web Rule Language (SWRL) [56] and N3Logic [57]. However, as opposed to N3Logic, SWRL does not support SNAF[12], a logical requirement for being used for validation. We thus use N3Logic. N3Logic supports at least OWL-RL inferencing [58], which can be included during validation.

The *rule language* introduced together with N3Logic is N3 [59]. N3 introduces an extension to the RDF 1.0 model, more specifically, it is a superset of Turtle [60]. N3 allows declaring inferencing rules, background knowledge, and constraints with the same language. As in RDF, blank nodes are understood as existentially quantified variables and the co-occurrence of two triples as in the RDF graph of formulas (11) and (12) is understood as their conjunction. More, N3 supports universally quantified variables, indicated by a leading question mark `?`.

```
?x :likes :IceCream.
```
(15)

stands for *"Everyone likes ice cream."*, or in first order logic

$$\forall x : \text{likes}(x, \text{ice-cream}) \tag{16}$$

[11] https://github.com/boschthomas/RDF-Constraints-Vocabulary

[12] https://github.com/protegeproject/swrlapi/wiki/SWRLLanguageFAQ#Does_SWRL_support_Negation_As_Failure

Rules are written using curly brackets `{ }` and the implication symbol `=>`. An `rdfs:subClassOf` relation such as `:Person rdfs:subClassOf :Researcher` can be expressed as:

```
{?x a :Researcher} => {?x a :Person}.
```
(17)

Moreover, the general `rdfs:subClassOf` relation for any class can also be expressed, as:

```
{?C rdfs:subClassOf ?D. ?X a ?C}
        => {?X a ?D}.
```
(18)

Reasoners that support N3Logic include FuXi, cwm, and EYE. FuXi[13] is a forward chaining production system for N3 whose reasoning is based on Rete algorithm [61]. The forward chaining cwm [17] reasoner is a general-purpose data processing tool which can be used for querying, checking, transforming and altering information. EYE [16] is a high-performance reasoner written in Prolog, enhanced with Euler path detection, allowing the creator of the rules to decide when to do forward reasoning and when backwards. EYE has generous support for built-in functions[14], among which, the RIF functions. For reasoning engine, we use the EYE reasoner [16] since it fulfills the expressiveness as presented in Section 4. Furthermore, its ability to combine forward and backward chaining proves especially useful since constraint types are mostly localized to single relationships [7]. This means backward chaining has a potentially large impact on the performance: reasoning during validation can be very targeted, and in most cases, only facts that are relevant to the defined constraints are inferred.

5.3. Implementation

Our implementation is dubbed "Validatrr": a validator using rule-based reasoning. A Node.js JavaScript framework was created to discover and retrieve the vocabularies and ontologies as required by the use case, manage the commandline arguments, etc. The implementation is available at https://github.com/IDLabResearch/validation-reasoning-framework, and the validation's set of rules is available at https://github.com/IDLabResearch/data-validation.

[13] http://code.google.com/p/fuxi/

[14] http://eulersharp.sourceforge.net/2003/03swap/eye-builtins.html

5.4. Execution example

As example, we validate an RDF graph with a custom entailment regime using SHACL constraints. We take into account the example of the introduction (formula (1)), but the case where `:Bob` has two birthdates defined. The implications of `rdfs:domain` (formula (2)) should be taken into account as defined in RDFS [6] during validation, and the SHACL constraint states that each person should have exactly one birthdate (Listing 1). The result should be in the SHACL validation report format. Using this example, we can detail every step as show in Fig. 2: The RDF graph with a specific entailment (*RDF graph**) and general constraint types (*Constraints**) are inferred using (custom) entailment regime rules (*Entailment regime*) and constraint translation rules (*Constraint translation*), after which the validation occurs (*Validation*), and the resulting violations are translated via rules (*Report*) in a specific report format (*Violations**).

```
:PersonShape a sh:NodeShape ;
  sh:targetClass :Person ;
  sh:property [
    sh:path :birthdate ;
    sh:minCount 1 ; sh:maxCount 1 ;
    sh:datatype xsd:date ] .
```

Listing 1: Person Shape in SHACL

To make sure `rdfs:domain` is correctly interpreted during validation, we include additional inferencing rules[15] (*Entailment regime*), described in N3 as

```
{?P rdfs:domain ?C. ?X ?P ?Y}
        => {?X a ?C} .
```
(19)

Given formula (19), it is inferred that `:Bob` is a person (*RDF graph**).

To make sure SHACL constraints are correctly interpreted, SHACL translation rules need to be included during validation (*Constraint translation*). The general "Exact Qualified Cardinality Restrictions" RDF-CV constraint is inferred from the SHACL constraint of Listing 1, using the rules of Listing 2 (*Constraints**).

```
{
  ?sh a sh:NodeShape ;
    sh:targetClass ?Class ;
```

[15] http://eulersharp.sourceforge.net/2003/03swap/rdfs-domain.html

```
    sh:property [
      sh:path ?p ;
      sh:minCount ?v ; sh:maxCount ?v1 ;
    sh:datatype ?C ] .
  ?v pred:numeric-equal ?v1
} => {
  ?constraint a rdfcv:SimpleConstraint ;
    :originalShape ?sh ;
    :constraintType :ExQualCardRestr ;
    rdfcv:constrainingElement
      :exact-cardinality ;
    rdfcv:contextClass ?Class ;
    rdfcv:leftProperties ?p ;
    rdfcv:classes ?C ;
    rdfcv:constrainingValue ?v
} .
```

Listing 2: Translate the SHACL shape to a general constraint type

Finding violations makes use of general rules, i.e., Listing 3 (*Validation*). Lines 11–14 define how to find a violation, relying on built-ins: gather a set of resources in a list (`e:findall`), calculate the length of that list (`e:length`), and mathematically compare numbers (`math:notEqualTo`). For all objects of a certain class or datatype related using predicate `?p` (e.g., `:birthdate`) where the number of objects is different from the constraint value `?v` (e.g., `1`), a violation is returned.

```
1  {
2    ?constraint a rdfcv:SimpleConstraint ;
3      :constraintType :ExQualCardRestr ;
4      rdfcv:constrainingElement
5        :exact-cardinality ;
6      rdfcv:contextClass ?Class ;
7      rdfcv:leftProperties ?p ;
8      rdfcv:classes ?C ;
9      rdfcv:constrainingValue ?v .
10   ?x a ?Class.
11   _:x e:findall
12     ( ?C {?x ?p ?o. ?o a ?C} ?list) .
13   ?list e:length ?l .
14   ?l math:notEqualTo ?v
15 } => {
16   _:v a :constraintViolation ;
17     :violatedConstraint ?constraint ;
18     :class ?Class ;
19     :instance ?x ;
20     :objectClass ?C ;
21     :property ?p
22 } .
```

Listing 3: Validate using general constraint types

The general violations are translated into a report format (Fig. 2, *Violations**), e.g., using the SHACL Validation Report [32] (see Listing 4). The result is a set of triples using the exact same input and output as a SHACL processor. However, the RDF graph's interpretation is explicit, the process is transparent, and custom inferencing can be included.

```
{
  _:v a :constraintViolation ;
    :violatedConstraint [
      :originalShape ?sh ;
      :constraintType :exact-cardinality
    ] ;
    :class ?Class ;
    :instance ?x ;
    :objectClass ?C ;
    :property ?p
} => {
  _:y a sh:ValidationReport ;
    sh:conforms false ;
    sh:result [
      a sh:ValidationResult ;
      sh:resultSeverity sh:Violation ;
      sh:focusNode ?x ;
      sh:resultPath ?p ;
      sh:resultMessage "No exact match" ;
      sh:sourceShape ?sh ]
} .
```

Listing 4: Translate the general violations to the SHACL validation report

Moreover, different constraint descriptions are easily supported via the general constraint types. Given the OWL restriction of Listing 5. Using a different set of rules, we can translate this restriction into the same constraint type (Listing 6). The validation process would continue exactly the same.

```
:Person rdfs:subClassOf _:x .
_:x a owl:Restriction ;
  owl:onProperty :birthdate ;
  owl:qualifiedCardinality
      "1"^^xsd:nonNegativeInteger ;
  owl:onDataRange xsd:date .
```

Listing 5: An OWL restriction

```
{
  ?Class rdfs:subClassOf ?c .
  ?c a owl:Restriction ;
    owl:onProperty ?x ;
```

```
    owl:qualifiedCardinality ?v ;
    owl:onDataRange ?C
} => {
  ?constraint a rdfcv:SimpleConstraint ;
    :originalShape _:x ;
    :constraintType :ExQualCardRestr ;
    rdfcv:constrainingElement
      :exact-cardinality ;
    rdfcv:contextClass ?Class ;
    rdfcv:leftProperties ?p ;
    rdfcv:classes ?C ;
    rdfcv:constrainingValue ?v
} .
```

Listing 6: Translate the OWL restriction to the general constraint type

6. Hypothesis validation

We validate the hypotheses of Section 1.2 comparing Validatrr against different approaches: Validatrr (i) explains accurately the root causes of why constraints are violated in more cases than the W3C SHACL recommendation, given the SHACL core constraint components (accepting H1, see Section 6.1); (ii) returns an accurate number of validation results with respect to the used (customizable) entailment regime, compared to an integrity constraints validator with a fixed entailment regime using RDFUnit (accepting H2, see Section 6.2); and (iii) supports more constraint types than other approaches (accepting H3, see Section 6.3). The performance evaluation of our implementation shows it is faster than the state of the art when combining inferencing and validation for commonly published datasets (accepting H4, see Section 6.4).

6.1. Root cause explanation of constraint violations

For each violation, Validatrr allows to determine either which part of the RDF graph is the root cause of the violation, or which axiom of the used ontology triggered an inference causing the violation. SHACL is a recent W3C Recommendation, standardizing how to describe the constraints and violations report for validating RDF graphs. We show Validatrr outperforms this W3C recommendation with respect to the root cause explanation of constraint violations, given the SHACL core constraint components.

The SHACL recommendation provides a set of test cases, enabling implementations prove compliance. The validation report denotes the violating resources, however, it is not possible to retrieve more information about the root cause. We revisit the previous example constraint that given a resource r, this resource has $(r_{firstname} \wedge r_{lastname}) \vee (r_{nickname})$[16]. The validation report when validating formula (1) using SHACL would be as in Listing 7. The SHACL validation report does not provide any details to identify why `:Bob` is invalid[17]. More, it is not possible to retrieve more information without inspecting the code base, given the validation approaches that support SHACL (either using a hard-coded system or a querying endpoint).

```
[ rdf:type sh:ValidationReport ;
  sh:conforms "false"^^xsd:boolean ;
  sh:result [
    rdf:type sh:ValidationResult ;
    sh:focusNode :Bob ;
    sh:resultSeverity sh:Violation ;
    sh:sourceConstraintComponent
      sh:OrConstraintComponent ;
    sh:sourceShape :PersonNameShape ;
    sh:value :Bob ; ] ; ]
```

Listing 7: Validation report of an OR constraint

Validatrr allows inclusion of the proof, showing the rules used to reach a conclusion. Listing 8 shows the proof's part containing the rules that calculate the actual violation. For `:firstname`, `:lastname`, and `:nickname`, we query objects that are linked using the respective predicate (Listing 8, lines 12–15, 18–21, and 24–27). $\mathcal{K}$ is the scope of our knowledge base, in which we look for violations. We count the number of objects found and compare them with the needed number. For `:firstname`, one linked object is found (Listing 8, lines 16–17), however, no linked object is found for `:lastname` nor `:nickname` (Listing 8, lines 22–23 and 28–29): a violation is returned.

```
1  <#lemma20> a r:Inference;
2  r:gives {
3  _:b1 a :constraintViolation.
4  _:b1 :violatedConstraint _:b2.
5  _:b1 :class :Man.
6  _:b1 :instance :Bob.
7  _:b1 :property :lastname.
8  _:b1 :property :nickname. };
```

[16] This is similar to the SHACL test case of https://github.com/w3c/data-shapes/blob/gh-pages/data-shapes-test-suite/tests/core/node/or-001.ttl

[17] https://www.w3.org/TR/shacl/#validator-OrConstraintComponent

```
 9  r:evidence (
10  ...
11  <#lemma37>
12  [ a r:Fact; r:gives { (𝒦 1) e:findall
13  (1
14  {:Bob :firstname _:b3}
15  (1))}]
16  [ a r:Fact; r:gives {(1) e:length 1}]
17  [ a r:Fact; r:gives {1 math:greaterThan 0}]
18  [ a r:Fact; r:gives {(𝒦 1) e:findall
19  (1
20  {:Bob :lastname _:b3}
21  ())}]
22  [ a r:Fact; r:gives {() e:length 0}]
23  [ a r:Fact; r:gives {0 math:lessThan 1}]
24  [ a r:Fact; r:gives {(𝒦 1) e:findall
25  (1
26  {:Bob :nickname _:b3}
27  ())}]
28  [ a r:Fact; r:gives {() e:length 0}]
29  [ a r:Fact; r:gives {0 math:lessThan 1}]).
```

Listing 8: Validation proof of an OR constraint

Validatrr can fully explain all root causes compared to 46% of SHACL-conforming approaches. Analysis of the SHACL specification showed that, out of the 28 core constraint components, 13 (46%) fully explain the root cause (summarized in Table 3). Eight of the other 15 components partially explain the root cause – e.g., a `sh:class` violation occurs when the targeted node is a literal, *or* is not classified accordingly, but this disjunction is not reflected in the validation report – and seven components do not explain the root cause at all – e.g., violations of nested shapes are not reflected in the validation report, only those of the top-level shapes. The SHACL validation report only returns which resource violates which constraint. The proof provided by Validatrr however gives insight into the exact workings of the validation, giving a detailed view of why a specific violation is thrown, and as the example shows, supports explanation of conjunction and nested shapes. We thus accept Hypothesis 1.

6.2. Unbiased number of validation results

Validatrr provides an accurate number of validation results without withholding functionality. To prove this, we first compare Validatrr with the state of the art functionally, and then include explicit semantics to clarify the difference and prove correct functioning.

Specifically, we compare with RDFUnit [3]. Hartmann et. al explicitly proposed using query-based approaches for validation [5], and RDFUnit is such a query-based approach, relying on a querying endpoint, and describing the constrains using SPARQL templates named Data Quality Test Patterns (DQTP). As such, RDFUnit is highly configurable and one of the implementations that supports SHACL[18].

Table 3
Analysis of root cause explanation of SHACL core constraint components. Validatrr can improve root explanations for 56% of the components compared to SHACL.

SHACL Name	Root Cause Explanation	Comment
`sh:class`	~	disjunction
`sh:datatype`	~	disjunction
`sh:nodeKind`	~	disjunction
`sh:minCount`	✗	no explanation
`sh:maxCount`	✗	no explanation
`sh:minExclusive`	✓	
`sh:minInclusive`	✓	
`sh:maxExclusive`	✓	
`sh:maxInclusive`	✓	
`sh:minLength`	~	disjunction
`sh:maxLength`	~	disjunction
`sh:pattern`	✓	
`sh:languageIn`	~	disjunction
`sh:uniqueLang`	✗	no explanation
`sh:equals`	✓	
`sh:disjoint`	✓	
`sh:lessThan`	✓	
`sh:lessThanOrEquals`	✓	
`sh:not`	✓	
`sh:and`	~	conjunction
`sh:or`	~	disjunction
`sh:xone`	✓	
`sh:node`	✗	nesting
`sh:property`	✗	nesting
`sh:qualifiedValueShape`, `sh:qualifiedMinCount`, `sh:qualifiedMaxCount`	✗	nesting
`sh:close`, `sh:ignoredProperties`	✓	
`sh:hasValue`	✓	
`sh:in`	✗	nesting

Functional comparison We compare with the original pattern library of RDFUnit [3]. This pattern library is the closest to the constraint types as introduced by Hartmann et al. [44]: the mapping between those two

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

is presented in previous work [47]. We test all unit tests defined by RDFUnit[19] after retrieving them as-is from the RDFUnit repository. As Validatrr validates general constraint types, a custom profile was created that translates the RDFUnit patterns to general constraint types. For a detailed explanation of the different test cases, we refer to the original RDFUnit paper [3].

The validation results depend on the used entailment regime. RDFUnit implicitly takes "every resource is an `rdfs:Resource`" and the `rdfs:subClassOf` construct into account, forming the custom entailment regime υ. We compare RDFUnit with Validatrr using three entailment regimes, taking into account no constructs at all ($\emptyset$), the custom (partially RDFS) entailment regime (υ), and full RDFS (ρ), respectively. Table 4 summarizes the results. For each constraint, we mention the test case's name, the number of violations that RDFUnit detects, and the number of violations that Validatrr detects using the different entailment regimes. The table shows the impact of using different entailment regimes (different amounts for violations are found for different entailment regimes for Validatrr), More, Validatrr can detect more violations under the same entailment regime (higher number of found violations for Validatrr under υ compared to RDFUnit).

Validatrr finds more violations and supports more constraint types than RDFUnit, denoted as starred constraints RDFSRANGE-MISS_wrong, RDFSRANGED_wrong, RDFSRANGE_correct, and RDFSRANGE_wrong. More violations are found, as RDFUnit does not yet support the constraint type *multiple ranges*: when a certain predicate is used, each resource linked as an object to that predicate should be classified into multiple classes. In all other cases, both solutions identify the same number of violations when using the same entailment regime. Validatrr thus functionally outperforms the pattern library (i.e., the corresponding constraint types) of RDFUnit (~H3).

Impact of including inferencing during validation Running Validatrr including different entailment regimes impacts the number of found violations. For Validatrr, we merely needed to include RDFS rules during validation. The results are found in Table 4 when comparing the different Validatrr columns. On the one hand, implicit violations are not found when no entailment regimes is included ($\emptyset$), as is the case for INVFUNC_wrong and OWLDISJC_wrong. On the other hand, violations are resolved early-on when including full RDFS entailment (ρ), as is the case for, e.g., RDFLANGSTRING_wrong. More, as all inferencing occurs during a single reasoning execution, the inferences' provenance is retained in the proof.

[19] https://github.com/AKSW/RDFUnit/tree/master/rdfunit-core/src/test/resources/org/aksw/rdfunit/validate/data

Table 4
Comparing Validatrr with RDFUnit using different entailment regimes ($\emptyset$, υ, and ρ) shows we can find more violations given the same used entailment regimes, and shows the impact of used entailment regimes. Constraints where Validatrr outperforms RDFUnit are starred. All differences with RDFUnit are marked gray.

Test Case	**# found violations**			
	RDFUnit	Validatrr		
	υ	$\emptyset$	υ	ρ
INVFUNC_correct	0	0	0	0
INVFUNC_wrong	2	0	2	2
OWLCARDT_correct	0	0	0	0
OWLCARDT_wrong_exact	6	6	6	6
OWLCARDT_wrong_max	2	2	2	2
OWLCARDT_wrong_min	2	2	2	2
OWLDISJC_correct	0	0	0	2
OWLDISJC_wrong	6	2	6	6
OWLQCARDT_correct	0	0	0	0
OWLQCARDT_wrong_exact	6	6	6	6
OWLQCARDT_wrong_max	2	2	2	2
OWLQCARDT_wrong_min	2	2	2	2
RDFLANGSTRING_correct	0	0	0	0
RDFLANGSTRING_wrong	2	2	2	0
*RDFSRANGE-MISS_wrong**	1	3	3	0
RDFSRANGED_correct	0	0	0	0
*RDFSRANGED_wrong**	2	3	3	0
*RDFSRANGE_correct**	0	5	4	0
*RDFSRANGE_wrong**	1	3	3	3
RDFSRANG_LIT_correct	0	0	0	0
RDFSRANG_LIT_wrong	3	3	3	1

Custom entailment regimes can be included during validation, we thus accept Hypothesis 2. This impacts the validation results, without loss of functionality (H3) or obscuring the root cause of a violation (H1).

6.3. More constraint types

Validatrr can support more constraint types than existing validation approaches RDFUnit, SHACL, and ShEx, whilst allowing to including inferencing during validation. In the previous section, we showed we functionally outperform the original pattern library of RDFUnit, and we can include inferencing during validation. In this section, we compare our number of supported

constraint to those of the current most prominent constraint languages, namely, SHACL [32] and ShEx [38].

We tested Validatrr against the general constraint types to prove it supports more types than SHACL and ShEx. We do not test specifically against the test cases of, e.g., SHACL, because Validatrr is independent of the constraint language. On https://github.com/IDLabResearch/data-validation, we provide a set of test cases that can be used to test different constraint types as presented by Hartmann et al. [44].

Hartmann et al. investigated the constraint type support of SHACL and ShEx, and state that their coverage is 52% and 30%, respectively [45]. Using a rule-based reasoning approach can potentially support all constraint types as the logical requirements are met. Validatrr, published on https://github.com/IDLabResearch/validation-reasoning-framework, can cover up to 94% – given the current expressive support for built-ins – and has been tested to cover a similar number of constraint types as SHACL[20]. After including the rules for the remaining constraint types, we functionally outperform SHACL and ShEx, accepting Hypothesis 3.

Achieving 100% coverage (i.e., the remaining five constraint types) requires additional development on the reasoner to support specific built-ins. "Whitespace Handling" and "HTML Handling" require parsing built-ins, and "Valid Identifiers" requires a built-in to test URIs' dereferencability. The remaining two types ("Structure" and "Data Model Consistency") are general constraint types, defined by Hartmann et al., requiring SPARQL support. Supporting these constraint types requires a translation from SPARQL queries to N3 rules, for which we refer to related work [62].

6.4. Speed

A declarative validation approach that supports custom inferencing is faster than a validation system that includes an inferencing preprocessing step. We first compare the performance of Validatrr to that of RDFUnit, both without and with employing an entailment regime. Finally, we make some concluding remarks with respect to RDF graph sizes.

For these performance evaluations, we used: 300 datasets with sizes ranging from ten to one million triples, and an executing machine consisting of 24 cores (Intel Xeon CPU E5-2620 v3 @ 2.40GHz) and 128GB RAM. All evaluations were performed using docker images for both approaches, the different tests were orchestrated using custom scripts. All timings include docker images' initialization time. The data is available at https://github.com/IDLabResearch/validation-benchmark/tree/master/data/validation-journal.

Performance comparison We compare the execution time of Validatrr with RDFUnit, following a similar methodology as performed in [3]. We compare for each schema the execution time for RDF graphs of varying sizes without any additional inferencing rules, as RDFUnit implements default constraints for a fixed set of schemas [3]. We consider six commonly used schemas: FOAF, GeoSPARQL, OWL, DC terms, SKOS, and Prov-O. The validated RDF graphs' size range from ten triples to one million triples, in logarithmic steps of base ten. Maximally ten different RDF graphs – per schema, per RDF graph size – were downloaded, by querying LODLaundromat's SPARQL endpoint [63].

We validate the different RDF graphs against their respective schema using the pattern library of RDFUnit, and measure total execution time of Validatrr and RDFUnit. The median execution time across all schemas is plotted against RDF graph size per approach (see Fig. 3). To make sure we can combine execution times across schemas, we accepted the null hypothesis that no significant difference in execution time was found between schemas, by performing an ANOVA statistical test with single factor "used schema" for measurement variable "execution time per triple", executed pairwise for all used schemas. The null hypothesis with $\alpha = 0.05$ was accepted for every pair. We do not plot the number of violations found, as statistical analysis shows no large correlation between execution time and number of found violations, neither for Validatrr or RDFUnit (-0.0203 and 0.0458, respectively).

Without inferencing, our implementation is already faster for small RDF graphs. We perform about an order of magnitude faster until 10,000 triples, namely, 1-2s per RDF graph compared to 30s per RDF graph. After 10,000 triples, the execution time of our implementation increases drastically, whereas it remains steady for RDFUnit: the set-up time of the query-based approach dominates the total execution time. When using a rule-based reasoning approach for validation, the number of triples that are taken into account are a decisive factor.

Validation complexity's performance impact We compare the execution time of Validatrr with RDFUnit when including an entailment regime. This way, we evaluate the impact of including additional inferencing

[20]The test report is available at https://github.com/IDLabResearch/validation-reasoning-framework/blob/master/reports/validatrr-rdfcv-earl.ttl

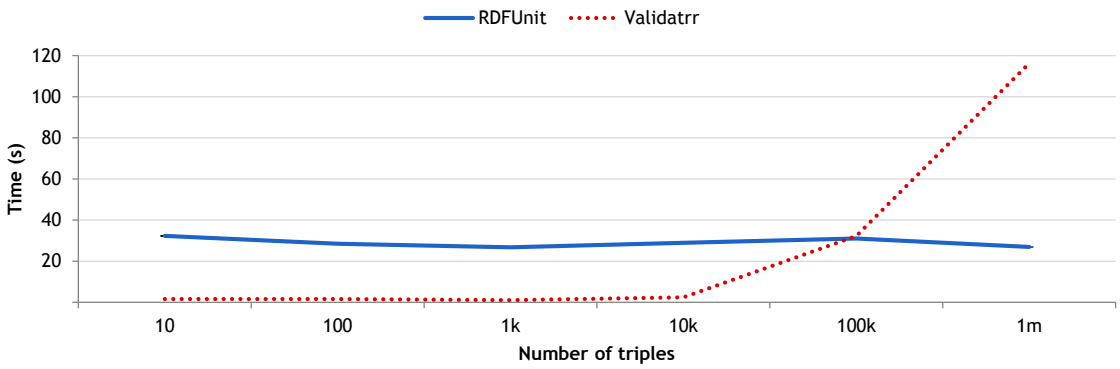


Figure 3. Validatrr executes up to an order of magnitude better when the number of triples per RDF graph is below 100,000 triples

rules during validation. For entailment regime, we use RDFS, as it is commonly used, and the evaluation of Section 6.2 already showed its effect on the number of violations found. For Validatrr, we could include the RDFS rules during validation. For RDFUnit, we included an RDFS entailment preprocessing step.

To make the results comparable, we used the EYE reasoner with the same RDFS rules to execute the reasoning preprocessing step. This also removes the need to compare with other entailment regimes than RDFS: the conclusions will be similar due to the usage of the same reasoner. For RDFUnit, the validation was executed on the newly inferred RDF graph, instead of the original RDF graph. The reasoning time and validation time are added to get the total execution time. The compared timings can be seen in Fig. 4.

Validatrr performs better with inferencing rules included than without, whereas the preprocessing step deteriorates RDFUnit's performance (P4). Until 10,000 triples execution time remains the same for both approaches, but for RDF graphs of one million triples, execution time drops from 120s to 80s for Validatrr, whereas it rises from 25s to 185s for RDFUnit. As most original violations are missing domain and range classes, this implicit data can be inferred. Where the query-based approach first needs to infer all implicit data before validation, Validatrr can infer this data during validation, needs to handle less violations, and thus performs better. We thus accept Hypothesis 4.

RDF graph size The evaluations show our approach and implementation provide more functionality with good performance for RDF graphs of 100,000 triples or less, with a scalability cost for large RDF graphs. To evaluate the scalability cost's impact, we investigate the average size of published RDF graphs and show that most published RDF graphs contain fewer than 100,000 triples.

We give an overview of the current state of published RDF graphs' size by analyzing the metadata provided by both LODLaundromat [63] and LODStats [64]. Both aim to provide a comprehensive picture of the current state of the LOD cloud, thus, their data is a reliable source to review the distribution of the RDF graph sizes of most currently published RDF graphs (Fig. 5). Both independent sources show a median of less than 100,000 triples per RDF graph.

For LODLaundromat, we queried all RDF graph sizes, and after selecting all distinct documents (i.e., ignoring the fragment identifier), we can conclude that 83% of all RDF graphs contain fewer than one million triples, and 60% contains at most 100,000 triples (see Fig. 5, mean 1,547,701 and median 41,290). The number of triples per RDF graph has a high positive skewness (24.711), which explains the large difference between mean and median, and guides us to plot the RDF graph sizes with logarithmic scale. Logarithmic transformation furthermore results in low kurtosis and skewness values – -0.102 and -0.572 respectively – allowing us to assume normal distribution.

LODStats provides the mean and median of the RDF graph sizes, which are 67,544 and 337 respectively. The difference between mean and median is large (thus further evidencing the need for a logarithmic scale), but the numbers are considerably lower than those of LODLaundromat. We queried the RDF graph sizes of LODStats (Fig. 5, dashed line), which shows even larger skewness towards small RDF graph sizes. Although we cannot assume normal distribution for LODStats, 94% of all RDF graphs in LODStats contain fewer than 100,000 triples, and 66% of currently published RDF graphs contain fewer than two thousand triples.

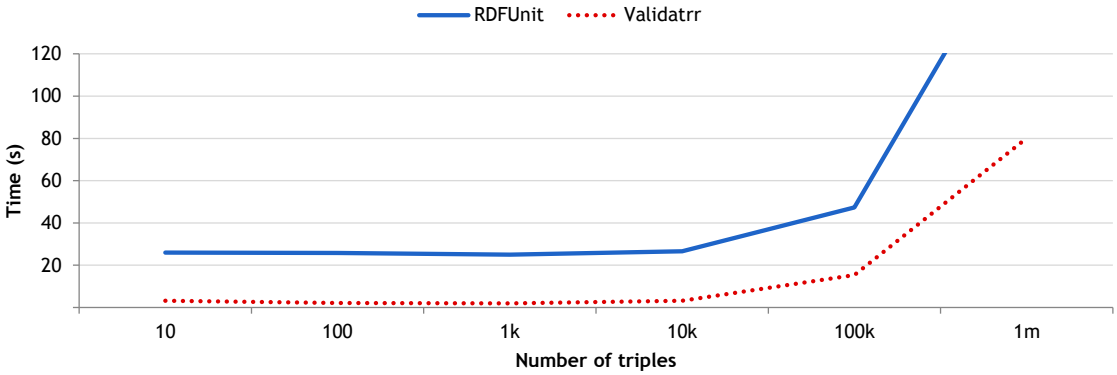


Figure 4. Validatrr scales generally better when including the RDFS inferencing rules. Complexity of the used schema has a large influence on the performance: validation taking the DBpedia ontology ("dbo") into account takes considerably longer time (x's denote processing times longer than 2 minutes)

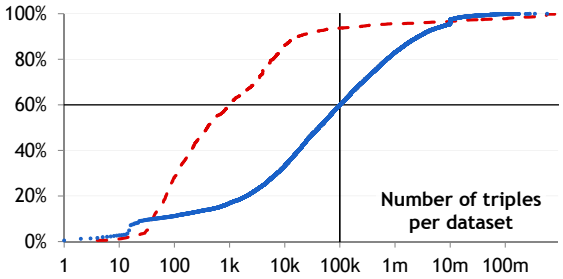


Figure 5. Cumulative percentage of RDF graphs by triple size (data points from LODLaundromat, dashed line from LODStats). From 60% to 94% of currently published RDF graphs contain fewer than 100,000 triples.

7. Conclusion and future work

A validation approach using rule-based reasoning gives more accurate validation results in terms of root causes, number of found violations, and constraint type coverage. Our corresponding implementation Validatrr provides these advantages and allows users to learn a single declaration and maintain a single implementation. More, Validatrr supports (custom) constraint languages and validation report descriptions. It performs better for RDF graphs up until 100,000 triples according to our evaluation. Further analysis showed that – depending on the source – this is case for 60–94% of currently published RDF graphs. Validatrr can thus be used in a variety of use cases where the functional validation requirements can be higher than what the current state of the art offers, without compromising on performance, for the majority of published RDF graphs.

This work paves the way for automatic RDF graph refinement, leading to higher quality RDF graphs faster. Our approach provides accurate descriptions of why a violation is returned. The returning violations are put into a richer context, making it easier for humans and machines to understand which steps should be taken to refine the RDF graph. Adding additional inferencing rules can provide machine-understandable suggestions to resolve violations. Taking these suggestions into account automatically improves the graph's adherence to the used schema or shape. RDF graphs can thus be refined without manual intervention.

The custom inferencing rules make our approach more versatile and applicable to more use cases and contexts. Namely, custom inferencing rules help validating *declarative RDF graph generation rules* automatically. The benefit of validating the generation rules instead of the resulting RDF graph – as these rules reflect how the RDF graph will be formed when generated – has been shown [4], however, this requires manual redefinition of the constraints. Custom entailment regimes can infer these redefinitions during validation, and only a single set of constraints needs to be maintained and understood.

This work can thus improve the generation, validation, and refinement process of RDF graphs, and increase the overall quality of the Semantic Web.

Acknowledgements

The described research activities were funded by Ghent University, imec, Flanders Innovation & Entrepreneurship (VLAIO), and the European Union. Ruben Verborgh is a postdoctoral fellow of the Research Foundation – Flanders (FWO).

References

[1] R. Cyganiak, D. Wood and M. Lanthaler, RDF 1.1 Concepts and Abstract Syntax, Technical Report, World Wide Web Consortium (W3C), 2014. http://www.w3.org/TR/rdf11-concepts/.

[2] J.M. Juran, *Juran's Quality Control Handbook*, 4th edn, Mcgraw-Hill, Texas, USA, 1988.

[3] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen and A. Zaveri, Test-driven evaluation of linked data quality, in: *Proceedings of the 23rd international conference on World Wide Web*, ACM, 2014, pp. 747–757.

[4] A. Dimou, D. Kontokostas, M. Freudenberg, R. Verborgh, J. Lehmann, E. Mannens, S. Hellmann and R. Van de Walle, Assessing and Refining Mappings to RDF to Improve Dataset Quality, in: *The Semantic Web – ISWC 2015*, M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan and S. Staab, eds, Lecture Notes in Computer Science, Vol. 9367, Springer International Publishing, Bethlehem, PA, USA, 2015, pp. 133–149.

[5] T. Bosch and K. Eckert, Requirements on RDF constraint formulation and validation, *International Conference on Dublin Core and Metadata Applications* (2014), 95–108, Citeseer.

[6] D. Brickley and R.V. Guha, RDF Schema 1.1, Recommendation, World Wide Web Consortium (W3C), 2014. http://www.w3.org/TR/rdf-schema/.

[7] T. Bosch, E. Acar, A. Nolle and K. Eckert, The role of reasoning for RDF validation, in: *Proceedings of the 11th International Conference on Semantic Systems*, ACM, 2015, pp. 33–40.

[8] A. Zaveri, A. Rula, A. Maurino, R. Pietrobon, J. Lehmann and S. Auer, Quality assessment for linked data: A survey, *Semantic Web Journal* **7**(1) (2015), 63–93.

[9] M.B. Ellefi, Z. Bellahsene, J. Breslin, E. Demidova, S. Dietze, J. Szymanski and K. Todorov, RDF Dataset Profiling - a Survey of Features, Methods, Vocabularies and Applications, *Semantic Web Journal* (2017).

[10] D. Tomaszuk, RDF Validation: A Brief Survey, in: *International Conference: Beyond Databases, Architectures and Structures*, Springer, 2017, pp. 344–355.

[11] C. Bizer and R. Cyganiak, Quality-driven information filtering using the WIQA policy framework, *Web Semantics: Science, Services and Agents on the World Wide Web* **7**(1) (2009), 1–10.

[12] K. Dentler, R. Cornet, A. ten Teije and N. de Keizer, Comparison of Reasoners for Large Ontologies in the OWL 2 EL Profile, *Semantic Web Journal* **2**(2) (2011), 71–87.

[13] P. Hitzler, M. Krötzsch, B. Parsia, P.F. Patel-Schneider and S. Rudolph, OWL 2 Web Ontology Language – Primer (Second Edition), Technical Report, World Wide Web Consortium (W3C), 2012. http://www.w3.org/TR/owl2-primer/.

[14] E. Prud'hommeaux et al., SPARQL 1.1 Overview, 2013, Accessed January 22nd, 2014.

[15] A. Paschke, Rules and Logic Programming for the Web, in: *Reasoning Web. Semantic Technologies for the Web of Data*, Springer Berlin Heidelberg, 2011, pp. 326–381.

[16] R. Verborgh and J. De Roo, Drawing Conclusions from Linked Data on the Web: The EYE Reasoner, *IEEE Software* **32**(5) (2015), 23–27.

[17] T. Berners-Lee, CWM, Technical Report, World Wide Web Consortium (W3C), 2000–2009. http://www.w3.org/2000/10/swap/doc/cwm.html.

[18] A. Hogan, A. Harth, A. Passant, S. Decker and A. Polleres, Weaving the Pedantic Web., in: *3rd International Workshop on Linked Data on the Web*, CEUR Workshop Proceedings, Vol. 628, CEUR, 2010.

[19] A. Hogan, J. Umbrich, A. Harth, R. Cyganiak, A. Polleres and S. Decker, An Empirical Survey of Linked Data Conformance, *Web Semant.* **14** (2012), 14–44.

[20] P.N. Mendes, H. Mühleisen and C. Bizer, Sieve: linked data quality assessment and fusion, in: *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, ACM, 2012, pp. 116–123.

[21] J. Tao, E. Sirin, J. Bao and D.L. McGuinness, Integrity Constraints in OWL., in: *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010)*, Atlanta, Georgia, USA, 2010.

[22] B. Motik, I. Horrocks and U. Sattler, Bridging the gap between OWL and relational databases, in: *Proceedings of WWW 2007*, Vol. 7, Elsevier, 2009, pp. 74–89.

[23] P.F. Patel-Schneider, Using Description Logics for RDF Constraint Checking and Closed-World Recognition, *Proceedings of the 29th AAAI Conference on Artificial Intelligence* (2014).

[24] M.A. Musen, The Protégé Project: A Look Back and a Look Forward, *AI Matters* **1**(4) (2015), 4–12.

[25] F. Chalub and A. Rademaker, Verifying Integrity Constraints of a RDF-based WordNet, in: *Global WordNet Conference*, 2016, p. 309.

[26] H. Pérez-Urbina, E. Sirin and K. Clark, Validating RDF with OWL Integrity Constraints, Technical Report, LLC, 2012. https://www.stardog.com/docs/4.1.3/icv/icv-specification.

[27] P.F. Patel-Schneider, Diverging Views of SHACL, Technical Report, Nuance Communications, 2017.

[28] M. Farid, A. Roatis, I.F. Ilyas, H.-F. Hoffmann and X. Chu, CLAMS: bringing quality to data lakes, in: *Proceedings of the 2016 International Conference on Management of Data*, ACM, 2016, pp. 2089–2092.

[29] H. Knublauch, J.A. Hendler and K. Idehen, SPIN – Overview and Motivation, W3C Member Submission, W3C, 2011. https://www.w3.org/Submission/spin-overview/.

[30] J.-E. Labra-Gayo, E. Prud'hommeaux, H. Solbrig and I. Boneva, Validating and describing linked data portals using shapes, *arXiv preprint arXiv:1701.08924* (2017).

[31] J. Debattista, S. Auer and C. Lange, Luzzu – A Methodology and Framework for Linked Data Quality Assessment, *J. Data and Information Quality* **8**(1) (2016), 4–1432.

[32] H. Knublauch and D. Kontokostas, Shapes Constraint Language (SHACL), W3C Recommendation, W3C, 2017. https://www.w3.org/TR/shacl/.

[33] K.B. Schiefer and G. Valentin, DB2 universal database performance tuning, *IEEE Data Eng. Bull.* **22**(2) (1999), 12–19.

[34] M. Nilsson, Description Set Profiles: A constraint language for Dublin Core Application Profiles, Working Draft, Dublin Core Metadata Initiative (DCMI), 2008. http://dublincore.org/documents/2008/03/31/dc-dsp/.

[35] T. Bosch and K. Eckert, Towards Description Set Profiles for RDF using SPARQL as Intermediate Language, *International Conference on Dublin Core and Metadata Applications* (2014), 129–137, Citeseer.

[36] A.G. Ryman, A. Le Hors and S. Speicher, OSLC Resource Shape: A language for defining constraints on Linked Data., *LDOW* **996** (2013).

[37] P.M. Fischer, G. Lausen, A. Schätzle and M. Schmidt, RDF Constraint Checking, in: *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT 2015)*, P.M. Fischer, G. Alonso, M. Arenas and F. Geerts, eds, CEUR Workshop Proceedings, Vol. 1330, CEUR-WS.org, Brussels, Belgium, 2015, pp. 2015–2012.

[38] E. Prud'hommeaux, J.E. Labra Gayo and H. Solbrig, Shape expressions: an RDF validation and transformation language, in: *Proceedings of the 10th International Conference on Semantic Systems*, ACM, 2014, pp. 32–40.

[39] S. Staworko, I. Boneva, J.E. Labra Gayo, S. Hym, E.G. Prud'hommeaux and H. Solbrig, Complexity and Expressiveness of ShEx for RDF, in: *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 31, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[40] I. Boneva, J.E. Labra Gayo and E.G. Prud'hommeaux, Semantics and Validation of Shapes Schemas for RDF, in: *The Semantic Web – ISWC 2017: 16th International Semantic Web Conference, Vienna, Austria, October 21–25, 2017, Proceedings, Part I*, Vol. 10587, C. d'Amato, M. Fernandez, V. Tamma, F. Lecue, P. Cudré-Mauroux, J. Sequeda, C. Lange and J. Heflin, eds, Springer International Publishing, Cham, 2017, pp. 104–120.

[41] J. Debattista, M. Dekkers, C. Guéret, D. Lee, N. Mihindukulasooriya and A. Zaveri, Data on the Web Best Practices: Data Quality Vocabulary, Working Group Note, World Wide Web Consortium, 2016. https://www.w3.org/TR/vocab-dqv/.

[42] F. Radulovic, N. Mihindukulasooriya, R. García-Castro and A. Gómez-Pérez, A comprehensive quality model for linked data, *Semantic Web* (2017), 1–22.

[43] B. Bozic, R. Brennan, K. Feeney and G. Mendel-Gleason, Describing Reasoning Results with RVO, the Reasoning Violations Ontology., in: *MEPDaW/LDQ@ ESWC*, 2016, pp. 62–69.

[44] T. Bosch, A. Nolle, E. Acar and K. Eckert, RDF Validation Requirements – Evaluation and Logical Underpinning, *arXiv preprint arXiv:1501.03933* (2015).

[45] T. Hartmann, Validation Framework for RDF-based Constraint Languages - PhD Thesis Appendix, Technical Report, Karlsruher Institut für Technologie (KIT), 2016. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000054062.

[46] P. Pauwels and S. Zhang, Semantic rule-checking for regulation compliance checking: an overview of strategies and approaches, in: *32rd international CIB W78 conference, Proceedings*, J. Beetz, L. van Berlo, T. Hartmann and R. Amor, eds, 2015.

[47] D. Arndt, B. De Meester, A. Dimou, R. Verborgh and E. Mannens, Using Rule Based Reasoning for RDF Validation, in: *RuleML+RR*, 2017.

[48] M. Arenas, S. Conca and J. Pérez, Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard, in: *WWW*, 2012.

[49] B. Parsia, N. Matentzoglu, R. Gonçalves, B. Glimm and A. Steigmiller, The OWL Reasoner Evaluation (ORE) 2015 Competition Report, in: *Proceedings of the 11th International Workshop on Scalable Semantic Web Knowledge Base Systems co-located with 14th International Semantic Web Conference (ISWC 2015)*, T. Liebig and A. Fokoue, eds, CEUR Workshop Proceedings, Vol. 1457, CEUR-WS.org, Bethlehem, PA, USA, 2015, pp. 2–15.

[50] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens and R. Van de Walle, RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data, in: *Proceedings of the 7th Workshop on Linked Data on the Web*, CEUR Workshop Proceedings, Vol. 1184, CEUR, 2014.

[51] C.V. Damásio, A. Analyti, G. Antoniou and G. Wagner, Supporting Open and Closed World Reasoning on the Web, in: *Principles and Practice of Semantic Web Reasoning*, Springer Berlin Heidelberg, 2006, pp. 149–163.

[52] M. Kifer, J. de Bruijn, H. Boley and D. Fensel, A Realistic Architecture for the Semantic Web, in: *Rules and Rule Markup Languages for the Semantic Web*, A. Adi, S. Stoutenburg and S. Tabet, eds, Lecture Notes in Computer Science, Vol. 3791, Springer Berlin Heidelberg, 2005, pp. 17–29.

[53] A. Polleres, C. Feier and A. Harth, Rules with Contextually Scoped Negation, in: *The Semantic Web: Research and Applications: 3rd European Semantic Web Conference, ESWC 2006 Budva, Montenegro, June 11-14, 2006 Proceedings*, Y. Sure and J. Domingue, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 332–347.

[54] M. Kifer, G. Lausen and J. Wu, Logical foundations of object-oriented and frame-based languages, *Journal of the ACM* **42**(4) (1995), 741–843.

[55] M. Kifer, Rule Interchange Format: The Framework, in: *RR 2008: Web Reasoning and Rule Systems*, D. Calvanese and G. Lausen, eds, Lecture Notes in Computer Science, Vol. 5341, Springer Berlin Heidelberg, 2008, pp. 1–11.

[56] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof and M. Dean, SWRL: A semantic web rule language combining OWL and RuleML, W3C Member Submission, W3C, 2004.

[57] T. Berners-Lee, D. Connolly, L. Kagal, Y. Scharf and J. Hendler, N3Logic: A logical framework for the World Wide Web, *Theory and Practice of Logic Programming* **8**(3) (2008), 249–269.

[58] D. Arndt, B. De Meester, P. Bonte, J. Schaballie, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, R. Van de Walle and E. Mannens, Improving OWL RL reasoning in N3 by using specialized rules, in: *Ontology Engineering: 12th International Experiences and Directions Workshop on OWL*, V. Tamma, M. Dragoni, R. Gonçalves and A. Ławrynowicz, eds, Lecture Notes in Computer Science, Vol. 9557, Springer, 2016, pp. 93–104.

[59] T. Berners-Lee, Notation 3 Logic, 2005.

[60] D. Beckter, T. Berners-Lee, E. Prud'hommeaux and G. Carothers, RDF 1.1 Turtle – Terse RDF Triple Language, Technical Report, World Wide Web Consortium (W3C), 2014. http://www.w3.org/TR/turtle/.

[61] C.L. Forgy, Rete: A fast algorithm for the many pattern/-many object pattern match problem, *Artificial Intelligence* **19**(1) (1982), 17–37.

[62] J.H. Soltren, Query-based database policy assurance using semantic web technologies, Master's thesis, Massachusetts Institute of Technology, 2009.

[63] W. Beek, L. Rietveld, H.R. Bazoobandi, J. Wielemaker and S. Schlobach, LOD Laundromat: A Uniform Way of Publishing Other People's Dirty Data, in: *Proceedings of the 13th International Semantic Web Conference*, Springer International Publishing, 2014, pp. 213–228, Springer.

[64] I. Ermilov, J. Lehmann, M. Martin and S. Auer, LODStats: The data web census dataset, in: *International Semantic Web Conference*, Springer International Publishing, 2016, pp. 38–46, Springer.