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 – DLab, Department of Electronics and Information Systems, Technologiepark-Zwijnaarde 122, 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. The correct functioning of Semantic Web applications requires that given RDF graphs adhere to an expected shape. This shape depends on the RDF graph and the application’s supported entailments of that graph. During validation, RDF graphs are assessed against sets of constraints, and found violations help refining the RDF graphs. However, existing validation approaches cannot always explain the root causes of violations (inhibiting refinement), and cannot fully match the entailments supported during validation with those supported by the application. These approaches cannot accurately validate RDF graphs, or combine multiple systems, deteriorating the validator’s performance. In this paper, we present an alternative validation approach using rule-based reasoning, capable of fully customizing the used inferencing steps. We compare to existing approaches, and present a formal ground and practical implementation “Validatrr”, based on N3Logic and the EYE reasoner. Our approach – supporting an equivalent number of constraint types compared to the state of the art – better explains the root cause of the violations due to the reasoner’s generated logical proof, and returns an accurate number of violations due to the customizable inferencing rule set. Performance evaluation shows that Validatrr is performant for smaller datasets, and scales linearly w.r.t. the RDF graph size. The detailed root cause explanations can guide future validation report description specifications, and the fine-grained level of configuration can be employed to support different constraint languages. This foundation allows further research into, a.o., handling recursion, validating RDF graphs based on their generation description, and providing automatic refinement suggestions.

Keywords: Constraints, Rule-based Reasoning, Validation

1. Introduction

Semantic Web data is represented using the Resource Description Framework (RDF), forming an RDF graph [20]. The quality of an RDF graph – its “fitness for use” [74] – heavily influences the results of a Semantic Web application [51]. An RDF graph’s fitness for use depends on its shape i.e., the RDF graph itself and the application’s supported entailments of that RDF graph. For example, some applications support inferring rdfs:subClassOf entailments [16], whereas other applications require the RDF graph to explicitly contain all classifying triples (i.e., rdfs:subClassOf entailment is not supported).

RDF graphs are validated by assessing their adherence to a set of constraints [50], and different applications (i.e., different use cases) specify different sets of constraints. Via validation, we discover (portions of) RDF graphs that do not conform to these constraints, i.e., the violations that occur. These violations guide the user to the resources and relationships related to the constraints. Refining these resources and relationships results in an RDF graph of higher quality [26], thus, RDF graph validation is an important element for the correct functioning of Semantic Web applications.

1.1. Validation problems

Let us consider the following example: an RDF graph containing people and their birthdates is validated. The use case dictates the set of constraints and the supported

*Corresponding author. E-mail: ben.demeester@ugent.be.
entailments. Specifically, we validate formula (1)\(^1\), with a relevant ontology represented in formula (2).

\[
\text{:Bob :firstname "Bob" ;} \quad (1)
\]

\[
:\text{birthdate } "1970-01-01" \text{xsd:date .} \quad (2)
\]

\[
:\text{birthdate rdfs:domain :Person .} \quad (3)
\]

\[
:\text{Bob a :Person .} \quad (3)
\]

Problem 1 (P1): finding the root causes of violations. For example, the use case dictates following compound constraint \(c_{\text{compound}}\): given a resource \(r\), this resource has \((\text{firstname} \land \text{lastname}) \lor (\text{nickname})\). When the RDF graph contains formula (1) and formula (3), :Bob should be marked as a resource violating \(c_{\text{compound}}\). However, when refining the RDF graph, the root cause of the violation is needed: does the resource lack firstname, lastname, or nickname?

For constraint types such as compound constraints, existing validation approaches typically return the resource that violates the constraint. However, more detailed descriptions are typically not provided, and manual inspection is needed to discover the root cause: why a resource violates a constraint. Without the root cause, it is hard to (automatically) refine the RDF graph and improve its quality.

Problem 2 (P2): the number of found violations depends on the supported entailments. A mismatch between which entailments are supported during validation and which entailments are supported by the use case influences, e.g., whether formula (3) is inferred or not. Thus, either too many or too few violations can be returned [13]. This difference in number of found violations gives a biased idea of the real quality of the validated RDF graph.

Too many violations: formula (2) specifies the domain of :birthdate. Let us validate that “every resource in the RDF graph that has a birthdate, is a person” given formula (1). When the entailments of formula (2) are not supported, this would result in a violation: formula (3) is missing in the RDF graph. However, when the entailments of formula (2) are supported, we can infer formula (3), and no violation is returned.

Too few violations: Let us validate that “every person in the RDF graph adheres to constraint \(c_{\text{compound}}\)” given formula (1). Formula (3) is not explicitly stated and the entailments of formula (2) are not supported. No violations are found: :Bob is not explicitly classified as a :Person, thus :Bob is not targeted by \(c_{\text{compound}}\). However, supporting those entailments can create new statements to be validated, and lead to new violations. For example, by inferring formula (3) using formula (2), :Bob is targeted by \(-\) and violates \(-\) \(c_{\text{compound}}\). Such violations are not found in the original RDF graph, but discovered due to the supported entailments.

Customizing the set of inferring steps during validation (e.g., whether \(\text{rdfs:domain}\) entailments are supported or not) allows to match the entailments supported by the use case with those of the validation approach. However, support for customizable inferring steps is limited. When a fixed set (or no set) of inferring steps is supported, a separate reasoning process is needed to infer unsupported entailments, and edge cases handling this fixed set cannot be validated accurately. For example, let us look at the W3C recommended Shapes Constraint Language (SHACL): a language for validating RDF graphs against a set of constraints [47]. SHACL specifies a fixed set of inferring steps during validation, namely, \(\text{rdfs:subClassOf}\) entailment. Thus, one cannot validate, e.g., whether an RDF graph explicitly contains all triples that link resources to all their classes given a set of \(\text{rdfs:subClassOf}\) axioms, as \(\text{rdfs:subClassOf}\) triples are always inferred by a conform SHACL validator. RDF graphs that do not contain all classifying triples will be valid according to SHACL validators, however, they are handled poorly by applications that do not support \(\text{rdfs:subClassOf}\) entailment.

Problem 3 (P3): Combining validation with a reasoning preprocessing step decreases performance. Entailments can be inferred by performing reasoning as a preprocessing step prior to validation [13], thus combining multiple systems. The resulting RDF graph then explicitly contains all supported entailments, given that the reasoner can be configured to only infer the entailments that are supported by the use case. The number of found violations is then accurate with respect to the use case (solving P2). However, this requires a sequence of independent systems. Thus, the preprocessing step possibly produces entailments not relevant for validation [13]. This independent generation of unnecessary entailments can decrease the performance compared to a single validation system. More, due to this sequence of independent systems, finding the root causes involves investigating the results of both systems: the

\(^1\)For the remainder of the paper, empty prefixes denote the fictional schema http://example.com/, other prefixes are conform with the results of https://prefix.cc.
validator who detects violations, and the reasoner who infers entailments.

1.2. Hypotheses

To solve aforementioned observed validation problems, we pose following hypotheses:

Hypothesis 1: root causes can be explained more accurately compared to existing validation approaches when using a declarative logic.

Hypothesis 2: a more accurate number of violations are found compared to existing validation approaches when supporting a custom set of inferencing steps.

Hypothesis 3: a validation approach supporting more accurate root cause explanations and a custom set of inferencing steps can support an equivalent number of constraint types compared to existing approaches.

Hypothesis 4: a validation approach supporting a custom set of inferencing steps is faster than an approach that includes the same inferencing as a preprocessing step.

1.3. Contributions

In this paper, we propose an approach for RDF graph validation that uses a rule-based reasoner as its underlying technology. Rule-based reasoners can generate a proof stating which rules were triggered for which returned violation. Thus, the root causes of violations can be accurately explained (solving P1).

A validation approach using rule-based reasoning natively support the inclusion of a custom set of inferencing steps by adding custom rules. The supported entailments during validation can thus be matched to the entailments supported by the use case, and the validation returns an accurate number of found violations (solving P2).

Moreover, rule-based reasoners only need a single language to declare both the constraints and the set of inferencing rules, and only a single system to execute the validation. Compared to a combination of a reasoner and a validation system, this approach does not lead to the generation of entailments unnecessary to the validation step, making it potentially faster than including an inferencing preprocessing step (solving P3).

Our contributions are as follows:

i An analysis of existing validation approaches and comparison to a rule-based reasoning approach.

ii A formal ground for using rule-based reasoning for validation.

iii An implementation using N3Logic [8] to define the inferencing and validation rules, executed using the EYE reasoner [73], supporting general constraint types as described by Hartmann et al. [35].

iv An evaluation of our approach, positioning it within the state of the art by functionally validating the hypotheses and comparing the validation speed.

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 by using a custom set of inferencing rules up to at least OWL-RL complexity and expressiveness; (c) the number of supported constraint types is equivalent to existing validation approaches; and (d) 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 give an overview of the state of the art (Section 2), after which we position and compare rule-based reasoning as validation approach (Section 3). We provide a formal ground (Section 4) and practical implementation (Section 5), evaluate our proposed approach (Section 6), and summarize our conclusions (Section 7).

2. State of the art

In this work, we propose an alternative validation approach using rule-based reasoning. 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. We conclude with an overview of general constraint types in Section 2.4, which allows us to functionally compare validation approaches. Our categorization is derived from the general quality surveys of Zaveri et al. [74], Ellefi et al. [28], and Tomaszuk [72], and from the “Validating RDF Data” book [51]. The related works 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 Data quality can be assessed by employing a set of data quality assessment metrics [9]. Quality
assumption 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 [74]. Validating an RDF graph directly relates to intrinsic quality dimensions, as defined by Zaveri et al. [74]: (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, i.e., the graph’s adherence to a certain schema or shape. In this paper, we specifically focus on RDF graph validation, i.e., the intrinsic dimensions.

Validation of an RDF graph can be automated by using a set of test cases, each assessing a specific constraint [50]. Violations of those constraints are then indicated when a validation returns negative results. Validation is typically achieved following Closed World Assumption (CWA): what is not known to be true must be false. For example, 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 :Human. Negative results are returned, indicating violations.

Reasoning Ontologies are prevalent in the Semantic Web community to represent the knowledge of a domain. Ontology languages are used to annotate asserted facts (axioms). Examples include RDF Schema (RDFS) [16] and the Web Ontology Language (OWL) [39]. Reasoning on top of these axioms is achieved, as the calculus of the used logic specifies a set of inferencing steps, inferring logical consequences (entailments) from these axioms [24]. Semantic Web logics – given the open nature of the Web – typically follow the Open World Assumption (OWA): what is not known to be true is simply unknown.

Semantic Web reasoners are typically description logic-based reasoners supporting OWL-DL or sub-profiles such as OWL-QL [55], or rule-based reasoners [59]. Description logic-based reasoners are typically optimized for specific description logics, such as KAON2 for SHI Q and FaCT++ for SROIQ. Rule-based reasoners typically follow two types of inferencing algorithms: forward chaining and backward chaining [59]. Whereas forward chaining tries to infer as much new information as possible, backward chaining is goal-driven: the reasoner starts with a list of goals and tries to verify whether there are statements and rules available that support any of these goals [59]. The employed rules define the logic followed by rule-based reasoners such as EYE [73] or cwm [6]. Whereas description logic-based reasoners have (optimized) inferencing steps for, e.g., rdfs:subClassOf and other RDFS or OWL constructs embedded, rule-based reasoners commonly rely on the general “implies” construct. Each rule specifies “A implies B”, where both the antecedent “A” and the consequence “B” can consist of statements [59]. Certain constructs such as rdfs:subClassOf can be translated into one or more rules.

There is a clear distinction between ontologies and the constraint set for RDF graph validation: ontologies focus on the representation of a domain, whereas RDF graph validation checks whether the resources of that graph conform to a desired schema [51]. It is not required that the representation of a domain aligns with the schema for validation. However, they can complement each other. The usage of ontologies prescribes a set of inferencing steps, for example, the FOAF ontology declares the rdfs:range of the foaf:knows predicate as foaf:Person [17]. Whether these inferencing steps are taken into account during validation or not, influences the number of found violations [13].

2.2. Validation Approaches

In this section, we discuss RDF graph validation approaches. Tools and surveys that cover quality dimensions other than the intrinsic dimensions such as accessibility or representational dimensions are out of scope. We discuss the approaches roughly in chronological order: hard-coded, using integrity constraints, query-based, and using a high-level language. Except from hard-coded systems, these validation approaches propose or use some kind of declarative means to describe RDF graph constraints.

2.2.1. Hard-coded

Hard-coded systems are a black box where the business logic lies within the code base: the implementation embeds both description and validation of constraints. Hogan et al. analyzed common quality problems both for publishing and intrinsic quality dimensions [40], providing an initial set of best practices [41]. Efforts focus on a limited set of configurable settings (i.e., turning constraint rules on or off) [53].
2.2.2. Integrity Constraints

For these validation approaches, the axioms of vocabularies and ontologies used by the validated RDF graph are interpreted as integrity constraints [54, 60, 71]. For example, disjointness forces a description logic-based reasoner to throw an error, which is interpreted as a violation. To combine CWA typically assumed for validation with OWA assumed in ontology languages, alternative semantics for these ontology languages are proposed. The underlying technology used is a description logic-based reasoner or a SPARQL endpoint.

Description logic-based reasoner Motik et al. [54] propose semantic redefinitions, where a certain subset of axioms are designated as constraints. To know which alternative semantics for OWL apply, constraints have to be marked as such. They propose to integrate their implementation with KAON2. Furthermore, custom integrity constraints for Wordnet have been verified using Protégé [56] with FaCT++ [18].

SPARQL endpoint Tao et al. [71] 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 SPARQL [1] for axioms described in RDF, RDFS, and OWL [71], 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 [60]. This implementation is incorporated into Stardog ICV [63]. Patel-Schneider separates validation into integrity constraints and Closed World recognition [60], showing that RDF and RDFS entailment can be implemented for both by translation to SPARQL queries.

2.2.3. Query-based

In query-based approaches, constraints are described and interpreted similar to SPARQL queries [61]: only RDF graphs whose structure is compatible with the defined structure are returned. These approaches use an embedded or external SPARQL endpoint as underlying technology.

CLAMS [29] 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. propose the SPARQL Infer-
allows to evaluate a part of the validated RDF graph using a custom function.

**SHACL** The Shapes Constraint Language (SHACL) [47] is the W3C Recommendation for validating RDF graphs against a set of constraints [47]. The core of SHACL is independent of SPARQL, which promotes the development of new algorithms and approaches to validate RDF graphs [52]. The original specification does not include a denotational semantics such as ShEx, however, the recent work of Corman et al. propose a concise formal semantics for SHACL’s core constraint components, and a way of handling recursion in combination with negation [19]. Advanced features of SHACL include SHACL Rules (to derive inferred triples from the validated RDF graph) and SHACL Functions (to evaluate a part of the validated RDF graph using a custom function) [49].

### 2.3. Validation reports

Validation reports handle identification of which data quality dimensions are assessed in general, and the representation of violations in particular.

To identify data quality dimensions, Radulvic et al. extended the Dataset Quality Ontology (daQ) [23] to include all data quality dimensions as identified by Záveri et al. [74], leading to the Data Quality Vocabulary [67]. This allows the comparison of data quality dimension coverage of different frameworks. The violations report itself allows to distribute and compare the violations found in an RDF graph, and can refer to the dimension specifications using aforementioned general vocabularies. For example, the Quality Problem Report Ontology assembles detailed quality reports for all data quality dimensions [22]. The Reasoning Violations Ontology (RVO) is used to represent integrity constraint violations [15], and Kontokostas et al [50] use the RDF Logging Ontology (RLOG) to describe RDFUnit’s violation results. Both ShEx and SHACL provide violation report descriptions, with means to specify the violating resources, using a ShapeMap [66] and a Focus node [47], respectively.

### 2.4. Constraint types

Hartmann né Bosch et. al identify eighty-one general constraint types [14]. These constraint types are an abstraction of specific constraints, independent of the constraint language used to describe them. 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 [16], 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:cardinality for a sh:targetSubjectsOf shape [47].

Moreover, Hartmann et al. provide a logical underpinning stating the requirements for a validation approach to support all constraint types [13]. For thirty-five out of eighty-one constraints types (43.2%), reasoning (up to OWL-DL expressiveness) can improve the validation: without reasoning, either too many or too few violations can be returned.

### 3. Comparative analysis

Different types of validation approaches are proposed in the state of the art. The most prominent approaches are hard-coded, based on integrity constraints, query-based, and using high-level languages. In this section, we compare them with our proposed rule-based reasoning approach. Our analysis is summarized in Table 1.

We adapt the framework presented by Pauwels et al. [62], which introduces comparative factors of key implementation strategies for compliance checking applications. We adjust these factors with respect to the validation problems identified in Section 1.1. We generalize the factors time, customization, and inferencing steps, and introduce explanation and reasoning preprocessing as validation-specific factors.

**Explanation** The explanation as to why a certain violation occurs (i.e., the root cause). The more specific a validator can explain, the easier it is to (automatically) refine the RDF graph and improve its quality. Existing approaches typically have the means to explain violations up to the level of which resource violates which constraint. Explanations of hard-coded approaches either need to be explicitly implemented, or are provided by inspecting the code base. When using integrity constraints, approaches exist for resolving inconsistencies. These approaches perform some sort of root cause analysis, but are usually targeted at refining the axioms of the ontologies themselves [33]. It is not a standard
feature to produce proofs of the results of description logic-based reasoners [58]. In a query-based approach, the used SPARQL endpoint returns bindings [1]. In the case of validation, it returns the violating resources, without additional explanation. High-level languages can have mechanisms to additionally include the violating resources in the validation report. For example, ShEx and SHACL provide ShapeMaps [66] and Focus nodes [47], respectively. SHACL’s Focus nodes can further specify which predicate and object cause the violation, except for, e.g., compound constraints. Using rule-based reasoning allows the generation of a logical proof, as rule-based reasoning relies on a general “implies” construct to describe rules, and rule-based reasoners typically do not contain description logic optimizations. Such a logical proof declares which rules were triggered to arrive at a certain conclusion, giving a precise explanation for the root causes of constraint violations. Where existing approaches typically have the means to explain violations up to the level of which resource violates which shape, a logical proof can provide a more detailed explanation.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Hard-coded</th>
<th>Integrity constraints</th>
<th>Query-based</th>
<th>High-level language</th>
<th>Rule-based reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explanation</td>
<td>No</td>
<td>Limited / Yes</td>
<td>Limited</td>
<td>Limited</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>Short*</td>
<td>Long</td>
<td>Long*</td>
<td>Short*</td>
<td>Long</td>
</tr>
<tr>
<td>Customization</td>
<td>Limited</td>
<td>Limited</td>
<td>Open</td>
<td>Open</td>
<td>Open</td>
</tr>
<tr>
<td>Inferencing steps</td>
<td>No / Limited</td>
<td></td>
<td>Yes*</td>
<td>Limited / Yes</td>
<td>Limited / Yes</td>
</tr>
<tr>
<td>Reasoning preprocessing</td>
<td>Yes</td>
<td>Limited</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table 1**

Comparing the prominent validation approaches with rule-based reasoning, using factors explanation, time, customization, inferencing steps, and reasoning preprocessing. The time row indicates which approaches’ execution time is influenced due to the reasoning preprocessing using an asterisk. The asterisk in the inferencing steps row indicates that approaches based on integrity constraints cannot combine with a custom set of inferencing steps that overlaps with the integrity constraints, as their semantics are redefined.

**Customization** The extent of customization each type of approach enables. Typically, ease of customization is improved by using a declarative language. Customization of a hard-coded system requires development effort, as the business logic is embedded within the code. Other approaches rely on 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. Description logic-based reasoners used to identify integrity constraints are typically optimized for description logics such as OWL-QL and OWL-DL. Customization is limited to the description logic that the reasoner is optimized for. Query-based approaches allow customization by defining additional SPARQL queries and registering custom functions [34]. Systems using high-level languages are customized using the declarations as specified by the used language. The adoption of ShEx and SHACL shows that these languages provide sufficient customization. The extension mechanisms of these languages such as Semantic Actions [66] and SHACL Advanced Features [49], respectively, allow to customize the validation even further.

**Inferencing steps** Whether the validation approach supports a (custom) set of inferencing steps. Hard-coded systems can support a fixed set of inferencing steps, but this set cannot be inspected or altered without investigating the code base. Approaches that use in-
tegrity constraints for validation propose alternative semantics of commonly agreed upon ontology languages to include, a.o., some form of CWA [54, 71]. This leads to ambiguity in the Semantic Web as an existing, globally agreed upon logic, is changed [3]. It is not possible to combine such validation with a (custom) set of inferencing steps within a description logic: the same inferencing step has different semantics whether it is used for validation or for inferring new statements.

SPARQL endpoints used for query-based approaches can support up to OWL-RL reasoning [46], or support up to RDF and RDFS entailment via translation of the SPARQL queries using property paths [71]. High-level languages such as SHACL allow specifying the used entailment regime [32]: SHACL validators may operate on RDF graphs that include entailments using the sh:entailment property [47]. Furthermore, SHACL Rules [49] can be used to a certain extent to generate inferred statements during validation. By design, rule-based reasoning allows inclusion of a set of additional (custom) inferencing rules [59]. Whereas existing approaches mostly allow configuration to support, e.g., a specific entailment regime, the customization of the set of inferencing steps is more fine-grained for rule-based reasoners. This can increase complexity, but also allows catering the validation to use cases that depend on a specific set of inferencing steps. The importance of such use cases is evidenced by the fact that SHACL Rules is proposed as an advanced feature to the SHACL specification [49].

Reasoning preprocessing Existing approaches have no support for including a custom set of inferencing steps, propose alternative semantics, or allow a specific entailment regime. By including a reasoning step as preprocessing step to these approaches (see Fig. 1.1), the entailments valid during validation can be matched with the entailments valid for the use case, even when that use cases requires a custom set of inferencing steps [13]. First, a reasoner – optionally, hence the dashed line – infers all valid entailments of the original RDF graph (Fig. 1.1, Reasoner), taking into account the axioms of the relevant ontologies and vocabularies (Axioms). Then, the newly generated RDF graph (RDF graph*) is validated with respect to the specified constraints (Fig. 1.1, Validator).

By using a preprocessed inferred RDF graph, multiple systems (i.e., the reasoner and the validator) need to be combined, configured, and maintained. This separates concerns, however, this also means that different languages may need to be learned and combined for specifying the inferencing steps and constraints. As these multiple systems are not aligned, the reasoner could infer a large number of new triples that are irrelevant to the defined constraints, which could lead to bad scaling (Fig. 1.1, RDF graph*). Also, explaining the violation is hindered. Even when the reasoner can differentiate between the original triples and the inferred triples, finding the root causes involves investigating the output of both systems: the validator detecting the violations, and the reasoner inferring the supported entailments.

Reasoning preprocessing is not required when using rule-based reasoning. The set of inferencing steps and the set of constraints can be defined using the same declaration (Fig. 1.2, Inferencing rules and Constraints *), and executed simultaneously on the RDF graph and the axioms. Which statements need to be inferred can be optimized guided by the set of constraints, and only the output of a single system needs to be investigated to explain the found violations.

4. Logical Requirements

In this section, we discuss the requirements for a logic to be a valid choice for RDF graph validation, and argue for using a rule-based logic.

Constraint languages need to cope with different constraint types depending on users’ needs. Each constraint type implies logical requirements. The constraint types and the requirements they entail are investigated by Hartmann et al., claiming that Closed World Assumption (CWA) and Unique Name Assumption (UNA) are crucial for validation [13]. These requirements are not common for Semantic Web logics, as data on the Web is decentralized, information is spread (“anyone can say anything about anything” [20]), and single resources can have multiple URIs. Instead, Semantic Web logics such as OWL-DL assume OWA and in general non-Unique Name Assumption [55]. Hartmann et al. emphasize the difference between reasoning and validation, and favor query-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 reasoning can be used for validation, even though typically CWA and UNA is not followed. Specifically, we state that the requirements for using rule-based reasoning are (i) supporting Scoped Negation as Failure (SNAF) [21, 45, 64] instead of CWA (Section 4.1),
(1) Preprocessing approach

(2) Rule-based reasoning approach

Figure 1. The preprocessing approach: first (optionally, hence the dashed line), a reasoner is used to generate intermediate data (RDF graph*). 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.

(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 [13, 60, 71]. Given that most Web logics assume OWA, this would require semantic redefinitions to include inferencing during validation [54], 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) is sufficient. This is an interpretation of logical negation: instead of stating that ρ does not hold (i.e., ¬ρ), it is stated that reasoning fails to infer ρ within a specific scope [21, 45, 64]. This scope needs to be explicitly stated. As such, SNAF keeps monotonicity.

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

:Kurt a :Researcher; (4)
:name "Kurt01". (5)

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:

∀x : (( x a :Researcher )∧ ¬( x a :Person ))→ (:constraint :isViolated "true". ) (6)

As stated, this constraint cannot be tested with OWA: the knowledge base contains the triple of formula (4), but not of:

:Kurt a :Person. (7)

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 (7).

This changes if we take into account SNAF. Suppose that K is the set of triples we can derive (either with or without reasoning) from our knowledge base of formulas (4) and (5). Having K at our disposal, we can test:

∀x : ((( x a :Researcher ) ∈ K)∧ ¬(( x a :Person ) ∈ K))→ (:constraint :isViolated. ) (8)

The second conjunct is not a simple negation, it is a negation with a certain scope, in this case K. If we add new data to our knowledge base, e.g., the triple of formula (7), we would have a different knowledge
base $K'$ for which other statements hold. The truth value of formula (8) would not change since this formula explicitly mentions $K$. 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 [13], i.e., every resource taken into account can only have one single name (a single URI in our case) [44]. 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 [13].

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

\[
\text{:Britain dbo:capital :London. (9)}
\]

\[
\text{:England dbo:capital :London. (10)}
\]

Since both :Britain and :England have :London as their capital and dbo:capital is an inverse functional property, an description logic-based reasoner would derive that

\[
\text{:Britain owl:sameAs :England. (11)}
\]

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

The related constraint – defined as INVFUNC by Kontokostas et al. [50] – 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 (9) and (10) 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 typically depends on the designers of a logic which features are supported, there are also common standards. One of them is the Rule Interchange Format (RIF), whose aim is to provide a formalism to exchange rules in the Web [43]. 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. (12)
```

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 ≠ for numerical values. In the example, the comparison would be true since 0 ≠ -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 present our approach that uses rule-based reasoning for validation. We discuss the different components and the workflow in Section 5.1, the underlying technologies in Section 5.2, and implementation in Section 5.3. We end with an example using rules in Section 5.4.
5.1. Customizable validation

Our validator consists of multiple components that can be configured by adjusting the different rule sets (Fig. 2). The execution is primarily handled using the rule-based reasoner as underlying technology.

The set of Inferencing rules specifies the supported entailments during validation. This set can either be a predefined set to support, e.g., RDFS entailment [16], or can be fully customized. Optionally, the relevant axioms are provided during validation. As such, the entailments supported by the use case can be matched during validation.

The set of rules forming the Constraint translation allows our validator to infer the general constraint types – common across existing constraint languages [14] – from specific constraint descriptions. It can thus infer these types from the constraints described in a specific language such as SHACL [47]. The general constraint types are described using RDF-CV\(^6\) [11], which generalizes the constraint types into a coherent structure. Our rule-based validator is thus constraint language-independent.

The set of rules forming the Validation allows our validator to infer violations on the RDF graph with all supported entailments, based on the general constraint types. This set of rules specifies how to detect each constraint type.

The set of rules forming the Report allows our validator to infer the resulting violations in the required format. This set can be adapted to, e.g., the SHACL report format [47].

As a result, this declarative approach is decoupled from ontology language, constraint language, and report format. When no additional rule sets are included (i.e., only the Validation rule set is used), this validator does not infer any entailments, only validates constraints described using RDF-CV, and returns a report in a format based on RDF-CV.

All rule sets and input data are taken into account during a single reasoner execution. As opposed to using a reasoning preprocessing step, the inferred entailments can be geared towards the specified constraints (when making use of a backward chaining reasoner), and no unnecessary entailments are produced. For example, when an axiom specifies the range of a certain path, but no constraints are related to that path, this range might not need to be inferred. Moreover, as you only have a single system, finding the root cause does not require investigation of multiple systems: the logical proof contains the complete overview of which rules were used to generate which entailments and which violations.

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) [42] and N3Logic [8]. However, as opposed to N3Logic, SWRL does not support SNAF\(^7\), a logical requirement for being used for validation. N3Logic supports at least

---
\(^6\)https://github.com/boschthomas/RDF-Constraints-Vocabulary
\(^7\)https://github.com/protegeproject/swrlapi/wiki/SWRLLanguageFAQ#Does_SWRL_support_Negation_As_Failure
The rule language introduced together with N3Logic is N3 [7]. Everything covered by RDF 1.1 Semantics [37] is covered in N3. Syntactically, it is a superset of Turtle [4], N3 allows declaring inferencing rules, axioms, and constraints in 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 (9) and (10) is understood as their conjunction. More, N3 supports universally quantified variables, indicated by a leading question mark ?.

\[ ?x :\text{likes} :\text{IceCream} \]. \hspace{1cm} (13)

stands for “Everyone likes ice cream.”, or in first order logic

\[ \forall x : \text{likes}(x, \text{ice-cream}) \]. \hspace{1cm} (14)

Rules are written using curly brackets { } and the implication symbol \( \Rightarrow \). An rdfs:subClassOf relation such as :Person rdfs:subClassOf :Researcher can be expressed as:

\[ \{ ?x a :\text{Researcher} \} \Rightarrow \{ ?x a :\text{Person} \}. \hspace{1cm} (15) \]

The general rdfs:subClassOf relation can be expressed as:

\[ \{ ?C \text{rdfs:subClassOf} ?D . \hspace{1mm} ?X a ?C \} \Rightarrow \{ ?X a ?D \}. \hspace{1cm} (16) \]

Reasoners that support N3Logic include FuXi, cwm, and EYE. FuXi\(^8\) is a forward chaining production system for N3 whose reasoning is based on the Rete algorithm [31]. The forward chaining cwm\(^6\) reasoner is a general-purpose data processing tool which can be used for querying, checking, transforming and altering information. EYE\(^9\) [73] 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\(^10\), among which, the RIF functions. We choose the EYE reasoner as it fulfills the requirements 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 [13]. 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/validatrr, and the set of validation rules (Fig. 2, center) is available at https://github.com/IDLabResearch/data-validation.

5.4. Execution example

As example, we validate an RDF graph with a custom set of inferencing steps 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 [16] 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 all supported entailments (RDF graph\(^*\)) and general constraint types (Constraints\(^*\)) are inferred using a (custom) set of inferencing rules (Inferencing rules) 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\(^*\)).

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

Listing 1: Person Shape in SHACL

\(^8\)http://code.google.com/p/fuxi/
\(^9\)https://github.com/josd/eye
\(^10\)http://eulersharp.sourceforge.net/2003/03swap/eye-builtins.html
To make sure `rdfs:domain` is correctly interpreted during validation, we include additional inferencing rules\(^{11}\) (Inferencing rules), described in N3 as

\[
\{ ?P \ rdfs:domain \ ?C. \ ?X \ ?P \ ?Y \} \Rightarrow \{ ?X \ a \ ?C \}. \quad (17)
\]

Given formula (17), 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*).

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

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

Validation 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` (in this case `:birthdate`) where the number of objects is different from the constraint value `?v` (in this case 1), a violation is returned (lines 16–21).

\[
\{ ?constraint \ a \ rdfcv:SimpleConstraint; \\
\quad :constraintType :ExQualCardRestr; \\
\quad rdfcv:constrainingElement :exact-cardinality; \\
\quad rdfcv:contextClass ?Class; \\
\quad rdfcv:leftProperties ?p; \\
\quad rdfcv:classes ?C; \\
\quad rdfcv:constrainingValue ?v
\} \Rightarrow \{ \\
\quad _:x \ e:findall \\
\quad ( ?C \ {?X \ ?p \ ?o. \ ?o \ a \ ?C} ?list). \\
\quad ?list \ e:length \ ?l. \\
\quad ?l \ math:notEqualTo \ ?v
\}
\]

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 [47] (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 supported entailments can be matched to the use case, and the process is a single reasoning execution with transparent rule sets.

\[
\{ _:v \ a \ :constraintViolation; \\
\quad :violatedConstraint [ \\
\quad \quad :originalShape \ ?sh; \\
\quad \quad :constraintType :exact-cardinality \\
\quad ]; \\
\quad :class \ ?Class; \\
\quad :instance \ ?x; \\
\quad :objectClass \ ?C; \\
\quad :property ?p
\}
\]

Listing 4: Generate a report format for violations.
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 continues exactly the same.

Listing 5: An OWL restriction

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

6. Hypothesis validation

To validate the hypotheses of Section 1.2, we compare Validatrr to different validation approaches. We show that Validatrr (i) accurately explains the root cause of why a violation occurs in more cases than specified in SHACL, given the SHACL core constraint components (accepting Hypothesis 1, see Section 6.1); (ii) returns an accurate number of validation results with respect to the used set of inferencing steps, compared to an integrity constraints validator with a fixed set of inferencing steps using RDFUnit (accepting Hypothesis 2, see Section 6.2); and (iii) supports an equivalent number of constraint types than existing approaches (accepting Hypothesis 3, see Section 6.3). The performance evaluation shows that our implementation is faster than the state of the art when combining inferencing and validation for commonly published datasets (accepting Hypothesis 4, see Section 6.4).

6.1. Root cause explanation of constraint violations

Using the logical proof, we increase the explanation’s accuracy compared to what is currently expected of a validation approach. SHACL is a W3C Recommendation standardizing the description of constraints and violation reports for RDF graph validation. We show that the logical proof produced by the rule-based reasoning execution provides more detailed root cause explanations of constraint violations, compared to SHACL’s violation report description.

The SHACL recommendation provides a set of test cases, enabling implementations prove compliance. The validation report denotes the violating resources via sh:focusNode, and in some cases can further specify the violating path via sh:resultPath and the violating value via sh:value. However, it is not always possible to retrieve such additional information about the root cause. We revisit the previous example constraint that given a resource \( r \), this resource has (\( r \text{firstname} \land r \text{lastname} \))\lor (\( r \text{nickname} \)). Validation of formula (1) using a conform SHACL implementation results in a validation report similar to Listing 7. The validation report does not provide any further details to explain why :Bob is invalid.


\[ \text{https://github.com/w3c/data-shapes/tree/gh-pages/data-shapes-test-suite/tests} \]

\[ \text{This example is similar to the following SHACL test case: https://github.com/w3c/data-shapes/blob/gh-pages/data-shapes-test-suite/tests/core/node/or-001.ttl} \]

\[ \text{https://www.w3.org/TR/shacl/#validator-OrConstraintComponent} \]
Due to this proof, Validatrr can provide detailed explanations for the root causes of violations for all SHACL core constraint components, compared to 46%–75% of SHACL-conforming implementations. Analysis of the SHACL specification shows that, out of the 28 core constraint components, 13 (46%) provide a full explanation of the root cause (summarized in Table 2). For eight of the remaining components (an additional 29%), the validation report returns which resource violates which constraint, but does not return a detailed explanation. For example, a sh:class violation occurs when the targeted node is a literal, or when the targeted node is not classified accordingly, but this disjunction is not reflected in the validation report. For the remaining seven components, the validation report does not provide an explanation at all. For example, violations of nested shapes are not reflected in the validation report, only violations of top-level shapes.

Compared to SHACL-conforming implementations, Validatrr supports, a.o., explanation of disjunction and nested shapes. Our approach provides detailed explanations for all core components of W3C’s recommended high-level language to describe constraints. We thus accept Hypothesis 1.

6.2. Accurate number of found violations

Validatrr finds a more accurate number of violations compared to the state of the art. To prove this, we first compare Validatrr with the state of the art functionally, and then include a set of inferencing steps to clarify the difference.

Specifically, we compare with RDFUnit [50]. Hartmann et. al explicitly proposed using query-based approaches for validation [11], and RDFUnit is such a query-based approach, relying on a SPARQL endpoint, and describing the constraints using SPARQL templates named Data Quality Test Patterns (DQTP). As such, RDFUnit is highly configurable and one of the implementations that supports SHACL.15

Functional comparison We compare with the original pattern library of RDFUnit [50]. This pattern library is the closest to the constraint types as introduced by Hartmann et al. [14]: the mapping between those two is presented in previous work [3]. We test all unit tests defined by RDFUnit16 after retrieving them as-is from the RDFUnit repository. As Validatrr validates


Listing 7: Validation report of an OR constraint

```xml
<#lemma20> a r:Inference;
2  r:give { 
3  _:_b1 a :constraintViolation.
4  _:_b1 :violatedConstraint _:_b2.
5  _:_b1 :class :Man.
6  _:_b2 :instance :Bob.
7  _:_b2 :property :lastname.
8  _:_b1 :property :nickname. }
9  r:evidence ( 
10  ... 
11  <#lemma37>
12  [ a r:Fact; r:give { (K 1) e:findall
13  (1
14  [Bob :firstname _:_b3]
15  (1)])]
16  [ a r:Fact; r:give { (1 e:length 1)]
17  [ a r:Fact; r:give { (1 math:greaterThan 0)]
18  [ a r:Fact; r:give { (K 1) e:findall
19  (1
20  [Bob :lastname _:_b3]
21  (0)]})]
22  [ a r:Fact; r:give { (0 e:length 0)]
23  [ a r:Fact; r:give { (0 math:lessThan 1)]
24  [ a r:Fact; r:give { (K 1) e:findall
25  (1
26  [Bob :nickname _:_b3]
27  (0)]})]
28  [ a r:Fact; r:give { (0 e:length 0)]
29  [ a r:Fact; r:give { (0 math:lessThan 1)]}).
```

Listing 8: Validation proof of an OR constraint
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 [50].

The validation results depend on the used set of inferencing steps. RDFUnit implicitly takes “every resource is an rdfs:Resource” and the rdfs:subClassOf construct into account, forming the custom set of inferencing steps υ. We compare RDFUnit with Validatrr using three sets of inferencing steps, taking into account (i) no entailment at all (∅), (ii) the custom set of inferencing steps (υ), and (iii) full RDFS entailment (ρ).

Table 2

Comparing RDFUnit to Validatrr using different sets of inferencing steps (∅, υ, and ρ). Validatrr finds more violations given the same set of inferencing steps, and the set of inferencing steps used impacts the result. Test cases where Validatrr outperforms RDFUnit are starred. Rows where Validatrr and RDFUnit differ are marked gray.

Table 3

Table 3 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 sets of inferencing steps. The table shows the impact of using different sets of inferencing steps: depending on the set, Validatrr finds a different number of violations. More, Validatrr detects more violations using the same set of inferencing steps: there is a 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 test cases rdfsrange-miss_wrong*, rdfsrange_correct*, rdfsrange_correct*, and rdfsrange_wrong*. 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 set of inferencing steps. Validatrr thus functionally out-

Table 2

<table>
<thead>
<tr>
<th>SHACL Name</th>
<th>Root Cause</th>
<th>Explanation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh:class</td>
<td>~</td>
<td>disjunction</td>
<td></td>
</tr>
<tr>
<td>sh:datatype</td>
<td>~</td>
<td>disjunction</td>
<td></td>
</tr>
<tr>
<td>sh:nodekind</td>
<td>~</td>
<td>disjunction</td>
<td></td>
</tr>
<tr>
<td>sh:minCount</td>
<td>✗</td>
<td>no explanation</td>
<td></td>
</tr>
<tr>
<td>sh:maxCount</td>
<td>✗</td>
<td>no explanation</td>
<td></td>
</tr>
<tr>
<td>sh:minExclusive</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:minInclusive</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:maxExclusive</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:maxLength</td>
<td>~</td>
<td>disjunction</td>
<td></td>
</tr>
<tr>
<td>sh:maxLength</td>
<td>~</td>
<td>disjunction</td>
<td></td>
</tr>
<tr>
<td>sh:pattern</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:languageIn</td>
<td>~</td>
<td>disjunction</td>
<td></td>
</tr>
<tr>
<td>sh:uniqueLang</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:equals</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:disjoint</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:lessThan</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:lessThanOrEquals</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:not</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:and</td>
<td>~</td>
<td>conjunction</td>
<td></td>
</tr>
<tr>
<td>sh:or</td>
<td>~</td>
<td>disjunction</td>
<td></td>
</tr>
<tr>
<td>sh:zone</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:node</td>
<td>✗</td>
<td>nesting</td>
<td></td>
</tr>
<tr>
<td>sh:property</td>
<td>✗</td>
<td>nesting</td>
<td></td>
</tr>
<tr>
<td>sh:qualifiedValueShape.</td>
<td>✗</td>
<td>nesting</td>
<td></td>
</tr>
<tr>
<td>sh:qualifiedMaxCount</td>
<td>✗</td>
<td>nesting</td>
<td></td>
</tr>
<tr>
<td>sh:close</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:ignoredProperties</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:hasValue</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh:in</td>
<td>✗</td>
<td>nesting</td>
<td></td>
</tr>
</tbody>
</table>
performs the pattern library (i.e., the corresponding constraint types) of RDFUnit.

Impact of including sets of inferencing steps during validation Running Validatrr using different sets of inferencing steps impacts the number of found violations. Validatrr is designed to easily configure this set using inferencing rules (Fig. 2, top-left). The results are found in Table 3, comparing the different Validatrr columns. On the one hand, certain violations are not found without entailment (∅), as is the case for INVFUNC_wrong and OWLDISJC_wrong. On the other hand, violations are resolved early-on when including RDFS entailment (ρ), as is the case for RDFLANGSTRING_wrong.

Compared to existing validation approaches, our approach allows including custom sets of inferencing steps during validation. The inferencing provenance is retained in the proof, as all inferencing occurs during a single reasoning execution. The logical proof can thus distinguish between violations that are caused due to constraint violations in the original RDF graph, or due to entailment during validation. We thus accept Hypothesis 2.

6.3. Equivalent number of constraint types

Validatrr can support an equivalent number of constraint types compared to existing validation approaches such as RDFUnit and SHACL. In the previous section, we showed we functionally outperform the original pattern library of RDFUnit whilst including a custom set of inferencing steps during validation. In this section, we compare our number of supported constraint types to that of SHACL [47].

We test Validatrr against general constraint types [36], to show that the number of supported constraint types is equivalent to SHACL. We do not test specifically against SHACL’s test cases, as Validatrr is independent of the constraint language. On https://github.com/IDLabResearch/data-validation, we provide a set of test cases, used to test these different constraint types.

Hartmann et al. investigated the constraint type support of SHACL, and stated that its coverage is 52% [36]. We updated the coverage report as presented by Hartmann et al. to take the latest SHACL specification and advanced features into account [47, 49]. The relevant data is available at Appendix A and online at https://github.com/IDLabResearch/constraint-types-coverage. This updated report shows that SHACL’s constraint type coverage is 84%.

Validatrr can cover up to 94% of all constraint types—given the current expressive support for built-ins—and has been tested to cover a similar number of constraint types as SHACL [17]. After including the rules for the remaining constraint types, we support an equivalent number of constraint types compared to SHACL. We thus accept 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 [69].

6.4. Speed

A validation approach that supports a custom set of inferencing steps is faster than a validation system that includes a reasoning preprocessing step. We first compare the performance of Validatrr to that of RDFUnit, both without and with a custom set of inferencing steps.

For these performance evaluations, we used 300 data sets 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 untampered docker images for both approaches to maintain reproducibility, the different tests were orchestrated using custom scripts. All timings include the 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 to RDFUnit, following RDFUnit’s original evaluation method. We use a default set of constraints for a fixed set of schemas, as defined by Kontokostas et al. [50]. We consider six commonly used schemas: FOAF, GeoSPARQL, OWL, DC terms, SKOS, and Prov-O. For each schema, we use RDF graphs of varying size. The validated RDF graphs’ size range from ten triples to one million triples, in logarithmic steps of base ten. At most ten different RDF graphs

17The test report is available at https://github.com/IDLabResearch/validatrr/blob/v0.2.0/reports/validatrr-rdfcv-earl.ttl
We validate the different RDF graphs against their respective schema using the default set of constraints and set of inferencing steps (υ) 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 in a log-log scale (see Fig. 3). To make sure we can combine execution times across schemas, we tested 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 α = 0.05 was accepted for every pair. The number of found violations are not plotted, 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).

Validatrr’s execution time is highly correlated with the number of triples of the validated RDF graph. Regression analysis shows an R square value of 0.9998, the null hypothesis with α = 0.05 is accepted: Validatrr’s execution time grows linearly with respect to the size of the validated RDF graph. Meanwhile, the execution time of RDFUnit remains constant at around 30s. This could largely be due to the set-up time required by RDFUnit, however, the timings attained via RDFUnit’s docker image does not allow us to draw further conclusions. The set-up time of RDFUnit thus possibly dominates the total execution time.

Without customizing the set of inferencing steps and docker images, Validatrr is faster for small RDF graphs. Validatrr is about an order of magnitude faster until 10,000 triples, namely, 1-2s per RDF graph compared to 30s per RDF graph for RDFUnit. After 100,000 triples, Validatrr is slower than RDFUnit, as Validatrr’s linearly growing execution time surpasses RDFUnit’s execution time.

Custom inferencing steps’ performance impact We compare the execution time of Validatrr to RDFUnit when using a custom set of inferencing steps. We use RDFS entailment (ρ): it is commonly used, and the evaluation of Section 6.2 showed it affects the number of violations found. For Validatrr, we include the RDFS rules during validation. For RDFUnit, we include an RDFS entailment preprocessing step, as RDFUnit’s docker image does not allow configuration to use a SPARQL engine that has inferencing capabilities. However, even if it would be possible to use a different SPARQL engine, a reasoning preprocessing step would still be needed for use cases that require support for a specific set of inferencing steps, not covered by typical entailment regimes [1].

To keep the measures comparable, we use the EYE reasoner as used in Validatrr with the same RDFS entailment rule set to execute the reasoning preprocessing step. This also precludes the need to compare with other sets of inferencing steps than RDFS entailment: the conclusions will be similar due to the usage of the same reasoner. Fig. 4 depicts the timings of RDFUnit and Validatrr. For RDFUnit, it depicts the combined timings of RDFS entailment as preprocessing step and validation on the newly inferred RDF graph(RDFUnit (ρ)), and it depicts solely the validation timings on the newly inferred graph (RDFUnit). For Validatrr, it depicts the timings of the validation with the two sets of inferencing rules (Validatrr (ρ) and Validatrr (υ), respectively).

Validatrr’s performance is not affected by using a different set of inferencing steps, whereas the preprocessing step deteriorates RDFUnit’s performance. This effect is noticeable starting from RDF graphs of 10,000 triples. For RDF graphs of one million triples, compared to the previous evaluation, median execution time rises from 27s to 210s for RDFUnit, largely due to the reasoning preprocessing step.

The number of found violations inversely affects the validation execution speed. Most original violations handle missing domain and range classes, which is inferred in RDFS entailment. Statistical analysis does not allow us to accept the null hypothesis that the number of violations found is inversely correlated to the execution time. However, we notice increased performance for both approaches when less violations need to be handled. Compared to previous evaluation, for one million triples, execution time (without reasoning preprocessing) drops from 27s to 21s for RDFUnit, and from 116s to 80s for Validatrr.

The performance evaluations show that the execution time of Validatrr outperforms RDFUnit for small RDF graphs up to 100,000 triples, and its linear scaling behavior is not affected by including RDFS entailment during validation. Validatrr outperforms RDFUnit when reasoning preprocessing is needed, i.e., when the used SPARQL endpoint does not support inferencing up to the needed expressiveness, or cannot be sufficiently customized to the use case. Where RDFUnit first needs to infer all implicit data before valida-
In this section, we discuss our proposed rule-based reasoning validation approach and introduced implementation. We provide concluding remarks and guide towards future work with respect to (i) the detailed root cause explanations, (ii) the fine-grained level of configuration, (iii) the number of constraint types supported by our approach, and (iv) the scaling behavior of Validatrr’s performance. We close by providing some further research perspectives.

The logical proof of a validation execution, generated by the rule-based reasoner, provides a more detailed root cause explanation of why a violation occurs than the state of the art. Our evaluation does not imply that existing approaches and implementations are not capable of providing a similar level of detail. However, it does show the feasibility of more detailed explanations, and the capability of our approach to generate them. To improve the level of detail of explanations provided in the validation report, our work can guide future iterations of, e.g., SHACL’s validation report descriptions, and the algorithms that generate them.

Our approach is fully configurable by adjusting different rule sets: only a single declaration and single implementation is needed to support different constraint languages, sets of inferencing steps, and validation report descriptions. This level of control considerably increases expressiveness and complexity of the validator, and a small change in a rule set could have large effects on the validation results. However, such fine-grained configuration is not needed for every use case. Future work requires investigation into configuration defaults for, a.o., ShEx and SHACL: to what extend can Validatrr be configured to function as a compliant ShEx or SHACL validator, and how will the combination of inferencing rule sets look like? A short-term
goal is showing that Validatrr with the right configuration passes the core SHACL tests and is included as a compliant SHACL validator in the respective W3C documentation. As such, we can provide a compliant SHACL validator where :entailment is accurately supported: the user can choose exactly which inferencing rule set is supported during validation, and can choose not to rely on the predefined custom set of inferencing steps (i.e., support for rdfs:subClassOf, but no other RDFS entailments) as currently specified in SHACL.

Our approach supports an equivalent number of constraint types compared to the state of the art, with description logic-expressiveness up to at least OWL-R1. An important point of interest is handling recursion, one of the main differences between ShEx and SHACL. The semantics of ShEx are defined, also for recursion [10], and – as it is currently undefined in the SHACL specification – current works are investigating recursion in combination with negation for SHACL. Future work for our approach is investigating recursion, taking into account the conclusions and mentioned complexity issues of aforementioned works. Accepting that the general problem is NP-Hard, using rule-based reasoning gives us a strong tool to handle recursion. A rule-based reasoner such as the EYE reasoner has path detection: different validations calling each other can be handled, as path detection prevents the reasoner from applying the same rule to the same data twice. In this regard, we can further investigate whether the strategies of Answer Set Programming [27] help to solve related problems, taking into account their two kinds of negation (Negation as Failure and strong negation). After investigating which rules are needed to handle recursion, the user can choose whether or not recursion should be supported during validation, as these extra rules can be added or not.

The performance of Validatrr is up to an order of magnitude faster than RDFUnit for RDF graphs up to 100,000 triples, and scales linearly w.r.t. the number of triples in the RDF graph. However, it scales less than RDFUnit, making Validatrr less suitable for large RDF graphs. As such, a trade-off must be made: our approach, which performs better for smaller RDF graphs, allows fine-grained configuration and detailed explanation, whereas other approaches scale better but do not provide the same level of detail. For future work, further investigation into related works that aim to improve the performance of rule-based reasoners, such as the work of Arndt et al. [2], can be used to improve the current scaling behavior of Validatrr.

Further research perspectives include validation of RDF graph generation descriptions, and automatic graph refinement based on violation explanations. The combination reduces the effort required to provide high-quality RDF graph generation descriptions, and is being further investigated by Heyvaert et al. [38].

On the one hand, a declarative description for generating an RDF graph – e.g., using the RDF Mapping Language (RML) [25] – can be validated, to show whether that description produces a valid RDF graph [26]. Certain constraints that apply to the description can be inferred based on the constraints that apply to the RDF graph. By including a custom inferencing rule set that reflects such inferencing in Validatrr, the generation description can be validated based on the set of constraints that apply to the RDF graph. As such, only a single set of constraints needs to be maintained and understood. The requirements of this custom inferencing rule set, and which constraint types can be applied to generation descriptions, is future work.

On the other hand, rules that handle the accurate explanations of why a violation is returned, can provide suggestions to (automatically) resolve the violation. For example, the constraint specifying “every book should have either an ISSN or an ISBN number” is violated by a resource that has both numbers. Suggestions include removing the ISSN number and removing the ISBN number. Which types of suggestions can be provided, and in which order these should be applied, is future work.

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).

Appendix A. Updated Constraint Types Coverage

### Table 4

Coverage of validation approaches w.r.t. constraint types. Taken from Hartmann et al. [36], updated take the recent advancements of SHACL into account [47, 49]. Changes w.r.t. the original tabe of Hartmann et al. are marked grey.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>SHACL</th>
<th>Validator</th>
</tr>
</thead>
<tbody>
<tr>
<td>A01</td>
<td>*Functional Properties</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A02</td>
<td>*Inverse-Functional Properties</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>A03</td>
<td>*Primary Key Properties</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>A04</td>
<td>*Subsumption</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A05</td>
<td>*Sub-Properties</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>A06</td>
<td>*Object Property Paths</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A07</td>
<td>Allowed Values</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A08</td>
<td>Not Allowed Values</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>A09</td>
<td>*Class Equivalence</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A10</td>
<td>*Equivalent Properties</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A11</td>
<td>Literal Value Comparison</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A12</td>
<td>Value is Valid for Datatype</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A13</td>
<td>*Property Domains</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A14</td>
<td>*Property Ranges</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A15</td>
<td>*Class-Specific</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A16</td>
<td>Data Property Facets</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A17</td>
<td>Literal Ranges</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A18</td>
<td>Negative Literal Ranges</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>A19</td>
<td>IRI Pattern Matching</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A20</td>
<td>Literal Pattern Matching</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A21</td>
<td>Negative</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>A22</td>
<td>*Existential Quantifications</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A23</td>
<td>*Universal Quantifications</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A24</td>
<td>*Value Restrictions</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A25</td>
<td>Use Sub-Super Relations</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Table 5

Coverage of validation approaches w.r.t. constraint types. Taken from Hartmann et al. [36], updated take the recent advancements of SHACL into account [47, 49]. Changes w.r.t. the original tabe of Hartmann et al. are marked grey (2).

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>SHACL</th>
<th>Validator</th>
</tr>
</thead>
<tbody>
<tr>
<td>A43</td>
<td>Repeatable Properties</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>A44</td>
<td>Conditional Properties</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A45</td>
<td>Recommended Properties</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A46</td>
<td>Severity Levels</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A47</td>
<td>Labeling and Documentation</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A48</td>
<td>Context-Sp. Property Groups</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A49</td>
<td>Context-Sp. Exclusive OR of P.</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A50</td>
<td>Context-Sp. Exclusive OR of P.</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A51</td>
<td>Context-Sp. Inclusive OR of P.</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A52</td>
<td>Context-Sp. Inclusive OR of P.</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A53</td>
<td>Mathematical Operations</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>A54</td>
<td>Ordering</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A55</td>
<td>*Inverse Object Properties</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A56</td>
<td>*Symmetric Object Properties</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A57</td>
<td>*Asymmetric Object Properties</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A58</td>
<td>*Transitive Object Properties</td>
<td>×</td>
<td>~</td>
</tr>
<tr>
<td>A59</td>
<td>*Self Restrictions</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A60</td>
<td>Valid Identifiers</td>
<td>×</td>
<td>X</td>
</tr>
<tr>
<td>A61</td>
<td>Recursive Queries</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A62</td>
<td>*Reflexive Object Properties</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A63</td>
<td>*Class-Sp. Reflexive Object P.</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A64</td>
<td>*Irreflexive Object Properties</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A65</td>
<td>*Class-Specific</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A66</td>
<td>Data Model Consistency</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>A67</td>
<td>Handle RDF Collections</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A68</td>
<td>Membership in</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A69</td>
<td>Disjoint Properties</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>A70</td>
<td>Disjoint Classes</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>A71</td>
<td>String Operations</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>A72</td>
<td>Aggregations</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A73</td>
<td>*Individual Equality</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A74</td>
<td>Individual Inequality</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A75</td>
<td>Context-Specific</td>
<td>X</td>
<td>~</td>
</tr>
<tr>
<td>A76</td>
<td>Valid Classes</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A77</td>
<td>Property Assertions</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>A78</td>
<td>*Intersection</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A79</td>
<td>*Disjunction</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>A80</td>
<td>*Negation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A81</td>
<td>*Default Values</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
References


