An Empirical Study on the "Usage of Not" in Real-World JSON Schema Documents

Mohamed-Amine Baazizi[1], Dario Colazzo[2], Giorgio Ghelli[3], Carlo Sartiani[4], and Stefanie Scherzinger[5]✉

[1] Sorbonne Université, LIP6 UMR 7606, France baazizi@ia.lip6.fr
[2] Université Paris-Dauphine, PSL Research University, France dario.colazzo@dauphine.fr
[3] Dipartimento di Informatica, Università di Pisa, Italy ghelli@di.unipi.it
[4] DIMIE, Università della Basilicata, Italy carlo.sartiani@unibas.it
[5] University of Passau, Passau, Germany stefanie.scherzinger@uni-passau.de

Abstract. We study the usage of negation in JSON Schema data modeling. Negation is a logical operator rarely present in type systems and schema description languages, since it complicates decision problems: many software tools, but also formal frameworks for working with JSON Schema, do not fully support negation. This motivates us to study whether negation is actually used in practice, for which aims, and whether it could — in principle — be replaced by simpler operators. We have collected a large corpus of 80k open source JSON Schema documents from GitHub. We perform a systematic analysis, quantify usage patterns of negation, and also qualitatively analyze schemas. We show that negation is indeed used, albeit infrequently, following a stable set of patterns.

Keywords: Empirical Study · Conceptual Modeling · JSON Schema.

1 Introduction

JavaScript Object Notation (JSON) has become one of the most popular formats for data exchange. While many schema languages for JSON have been proposed [3], JSON Schema [16] is receiving considerable attention. The theoretical properties of this language have been recently studied [4,7,17]. In this language, a schema is a logical combination of assertions, describing classes of constraints on objects, arrays, and base values. JSON Schema is constantly evolving and new drafts always introduce new features. The language is increasingly used for defining *domain-specific* data exchange formats [13] and as a meta-language for defining other languages; a subset of JSON Schema serves as the schema language inside MongoDB [15]. As a consequence, an active and quite broad development community is releasing JSON Schema tools (validators [1], in particular).

JSON Schema is powerful but complex, and its semantics is based on an intricate interplay among logical assertions. A distinctive feature is the `not` operator, whereby negation can be applied to any assertion. Negation is quite rare in type and schema languages, as it poses severe challenges.

(a)
```
{ "not":
  { "required": ["DisplaceModules"] }
}
```

(b)
```
{ "description": "...",
  "@errorMessages":
    { "not": "Invalid target: ..." },
  "not": {  "pattern": "..." } ... }
```

(c)
```
{ "title" : "Object w/ required foo.",
  "type": "object",
  "properties": {
    "foo": { "type": "integer" },
    "bar": { "type": "string" } },
  "patternProperties": {
    "f.*o": { "type": "integer" } },
  "required": ["foo"]
}
```

Fig. 1. Snippets of JSON Schema documents.

Example 1. One usage of `not` that startles novices (as discussed on StackOverflow [18]) is in combination with the keyword `required`, as shown in Figure 1(a). While "not required" may sound like "optional", it enforces that the object must violate the assertion, so member `"DisplaceModules"` must be *absent*.

Indeed, the `not`-operator is often not fully supported, whether in academic prototype tools [10], commercial tools (e.g., [15]), or even formal frameworks [11]. This inspired us to investigate the usage of this operator in real-world schemas, in a principled analysis of 80k JSON Schema documents crawled from GitHub. We formulate these research questions: (1) *how frequent* is negation in practice, (2) *how* is negation used, and (3) *what* are common usage patterns?

Contributions. We summarize the highlights of our systematic empirical study on the usage of `not`, which we describe in full detail in our extended technical report [5]. Along our journey towards understanding `not`, we gained a general understanding of JSON Schema modeling in practice. In particular:

- We establish a method for the collection and preparation of JSON Schema documents, and we make our corpus of schemas available (https://doi.org/10.5281/zenodo.5141199), as well as a docker container populated with data and pre-defined pattern queries for interactive, ad-hoc analysis in follow-up studies (https://doi.org/10.5281/zenodo.5141378).
- We measure the frequency of use of JSON Schema operators and of paths that include `not`, and quantify main patterns of use.
- We identify well-supported *jargons*, i.e., common uses of `not` that have the potential to mature into JSON Schema *design patterns*.

2 Preliminaries

JSON data model. The grammar below captures the syntax of JSON values, which are basic values, objects, or arrays. Basic values B include the null value, booleans, numbers n, and strings s. Objects O represent sets of members, each member being a name-value pair, and arrays A represent sequences of values.

$J ::= B \mid O \mid A$		**JSON expressions**
$B ::=$ null $\mid$ true $\mid$ false $\mid n \mid s$	$n \in \mathsf{Num}, s \in \mathsf{Str}$	**Basic values**
$O ::= \{l_1 : J_1, \ldots, l_n : J_n\}$	$n \geq 0, \quad i \neq j \Rightarrow l_i \neq l_j$	**Objects**
$A ::= [J_1, \ldots, J_n]$	$n \geq 0$	**Arrays**

JSON Schema. JSON Schema is a language for defining the structure of JSON documents. JSON Schema uses JSON syntax, its semantics has been formalized in [17] (following Draft-04). We limit ourselves to discussing the main keywords, and continue with two illustrative examples:

Assertions include `required`, `enum`, `const`, `pattern` and `type`, and indicate a test that is performed on the corresponding instance.

Applicators include the boolean operators `anyOf`, `allOf`, `oneOf`, `not`, the object operators `properties`, `patternProperties`, `additionalProperties`, the array operator `items`, and the reference operators `$ref`. Applicators indicate a request to apply a different operator to the same instance or to a component of the current instance.

Annotations include `title`, `description`, and `$comment`, they do not affect validation, but they indicate an annotation that should be associated with the instance. Since we are mostly interested in validation, and since, moreover, annotations are removed by the `not` operator, we will ignore them.

Example 2. In the schema in Figure 1(c), inspired from [1], line 1 carries an annotation. In defining an object (line 2), applicators define constraints on properties (lines 3), and the type of the properties matching a pattern (see line 6). Using an assertion, it is possible to indicate `required` properties (line 8).

Example 3. JSON Schema is an open standard: In Figure 1(b), `@errorMessages` is a user-defined keyword whose value is an object that describes the error, and not a JSON Schema assertion (link to schema 32451 in our schema corpus, available in the PDF). Hence, `not` in line 3 is just a member name, whereas negation does occur in line 4. The same string token has different semantics, depending on its context, which complicates parsing.

2.1 Pattern Queries

To study which keywords occur below an instance of the `not` operator, we introduce a simple path language. A path such as `.**.not.required` matches any path that ends with an object field named `required` found inside an object field whose name is `not`. Paths are expressed using the following language. Path matching is defined as in JSONPath [9].

$$p ::= step \mid step\ p \qquad step ::= .key \mid .* \mid [*] \mid .**$$

The step $.*$ retrieves all member values of an object, $[*]$ retrieves all items of an array, and $.**$ is the reflexive and transitive closure of the union of $.*$ and $[*]$, navigating to all nodes of the JSON tree to which it is applied.

Complex sub-schemas. We say that `not` has a *complex* sub-schema, when its object argument contains more than one keyword. In this case, we say these keywords *co-occur* in the negated schema; otherwise, a sub-schema is *simple*. As an example, consider the schema of Figure 3(b) (link to schema 3460): the argument of `not` is complex, and we match the paths `.not.enum` and `.not.type`.

3 Methodology

Context. We used the cloud service Google BigQuery to search for open source JSON Schema documents (excluding the schemas defining the JSON Schema drafts) on GitHub. We identified 91,6k URLs in July 2020, of which 85,6k could be retrieved (using `wget`). Discarding files with invalid syntax yields 82k files.

For each retrieved file, we analyzed the `$schema` declarations to identify the version of JSON Schema. Draft 2019-09 (also known as Draft-08) is still quite new, and not really represented. Draft-04 is declared in the vast majority of the files (79%), while Draft-07, Draft-06, and the old Draft-03 are each below 5%. An analysis of the file contents showed that the actual version that a schema follows is often different from the version declared.

Data Preparation. As a first data preparation step, we renamed all references (`$ref`) by a new keyword `$eref`, with the target of the reference as its child. Note that we did not expand references recursively. We expanded references to external documents, provided that we were able to locate the referenced document (e.g., either contained within our corpus, or by downloading the document). References were renamed to `$fref` when expansion failed. We observed that by expanding references we lose the conceptual information encoded in the reference path itself. Thus, `$ref` is often more than just a syntactic macro.

The schema corpus contains a large share of near-duplicate schemas, with small variations in syntax. We performed duplicate elimination by comparing compact *schema signatures*, defined as a function that maps each JSON Schema keyword to the number of its occurrences in the schema (encoded as a vector of keyword counts); we assumed that two schemas with the same *signature* are, with high probability, versions of the same schema, and we retained just one.

As illustrated in Example 3, correctly recognizing keywords can be a challenge. For this reason, we renamed all property names to avoid confusion when searching for patterns that involve the keyword `not`. As schema authors can define their own keywords, we have no way to know whether their value should be interpreted as an assertion. We experimented with two approaches: a "strict" approach in which we renamed everything that was inside a user-defined keyword, hence making it inaccessible by the analysis, and a "lax" approach in which we kept the content of any user-defined keyword, so that all instances of `not` in Figure 1(b) would be counted as keywords. With the strict approach, some interesting usage patterns are lost, and keyword usage is under-estimated. With the lax approach, we risk "false positives", and hence over-estimation. We decided that the over-estimation of the lax approach was preferable.

Analysis Process. The bulk of our effort is actually invested in data preparation. After experimenting with different data analysis platforms, we resorted to a relational encoding of the JSON Schema documents in PostgreSQL. This setup met our performance expectations, and allowed us to write queries in plain SQL.

Artifact Availability. Our schema corpus, as well as a docker image with our data analysis setup, are available on Zenodo (see the DOIs linked in the Introduction).

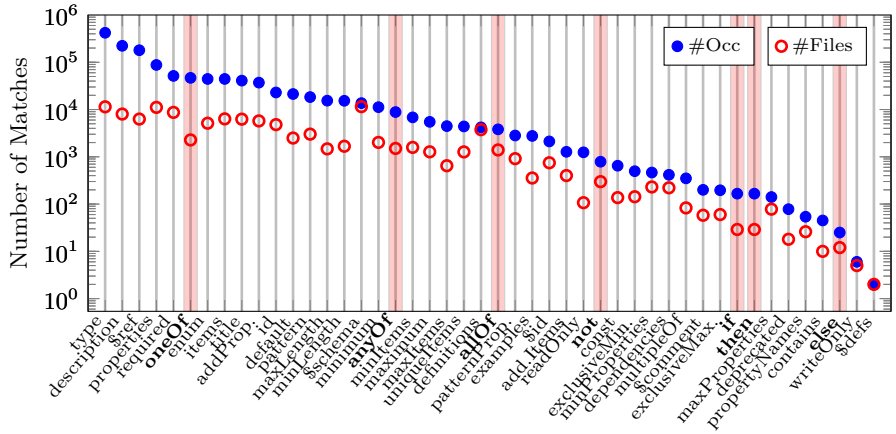

Fig. 2. Number of total occurrences (#Occ), and number of files (#Files), where a JSON Schema keyword appears. Boolean operators are highlighted.

4 Results of the Study

4.1 RQ1: How frequent is negation in practice?

We study the frequency of JSON Schema keywords within our corpus, and the Boolean operators (among them, negation). The reported absolute values are mainly interesting as indicators as to the relative occurrences of operators. Figure 2 visualizes the results. From left-to-right, we sort keywords by their number of occurrence (note the log-scaled vertical axes). We also show the number of files in which keywords occur, as a further indicator of keyword relevance.

The operator `not` appears in approx. 3% of all schemas, and occupies the 30th position, out of 46 keywords analyzed. Thus, it is a comparatively rare operator. The most common Boolean operator is `oneOf`, more frequent than `anyOf`. `allOf` is even less common. The Boolean operator `if-then-else` is even less common than `not`, but was only been introduced in Draft-07.

Results. We found the dissemination of `oneOf` surprising, since the exclusive-disjunctive semantics of `oneOf` is more complicated than the purely disjunctive `anyOf`: `oneOf` takes as argument a collection of subschemas $S_1, \ldots, S_n$, and a value J satisfies `oneOf` only if it matches exactly one subschema; `anyOf` is satisfied by any value J that matches at least one of the subschemas. Our hypothesis is that the description of a class as a `oneOf`-combination of a set of "subclasses" is familiar from the exclusive-subclassing mechanism of object-oriented languages.

The operator `not` appears 787 times in 298 different files out of 11,500. While not very frequent, its usage nevertheless merits a systematic study.

4.2 RQ2: How is negation used in practice?

We evaluated pattern queries to identify keywords below `not`. Table 1 summarizes the results. Consider the left half. We match the path `.**.not.*` 840 times

Table 1. Occurrences of `not.`k paths (overall #Occ, and counting #Files).

Path	#Occ	#Files	Path	#Occ	#Files
`not.*`	840	289	`not.$eref.*`	338	28
`required`	28.6 %	29.1 %	`required`	10.7 %	53.6 %
`items`	15.0 %	9.3 %	`items`	0.0 %	0.0 %
`type`	7.4 %	17.7 %	`type`	15.1 %	71.4 %
`properties`	8.5 %	16.3 %	`properties`	11.8 %	64.3 %
`$eref`	11.1 %	9.7 %	`$eref`	0.0 %	0.0 %
`enum`	7.3 %	18.0 %	`enum`	3.6 %	28.6 %
`allOf`	2.7 %	8.0 %	`allOf`	11.2 %	17.9 %
`pattern`	5.6 %	9.7 %	`pattern`	0.0 %	0.0 %
`anyOf`	5.4 %	12.5 %	`anyOf`	0.6 %	7.1 %
`description`	0.5 %	1.4 %	`description`	12.1 %	25.0 %
`title`	0.2 %	0.7 %	`title`	11.5 %	25.0 %
`$schema`	0.0 %	0.0 %	`$schema`	12.1 %	32.1 %
`$fref`	3.2 %	4.8 %	`$fref`	0.0 %	0.0 %
`oneOf`	0.7 %	1.4 %	`oneOf`	5.3 %	10.7 %
`additionalProperties`	1.3 %	3.8 %	`additionalProperties`	2.7 %	25.0 %
`patternProperties`	1.8 %	5.2 %	`patternProperties`	0.0 %	0.0 %
`const`	0.7 %	0.4 %	`const`	0.0 %	0.0 %
`definitions`	0.0 %	0.0 %	`definitions`	0.9 %	10.7 %
`id`	0.0 %	0.0 %	`id`	0.6 %	7.1 %
`dependencies`	0.0 %	0.0 %	`dependencies`	0.6 %	7.1 %
`not`	0.0 %	0.0 %	`not`	0.6 %	7.1 %
`$ref`	0.0 %	0.0 %	`$ref`	0.6 %	7.1 %
`$comment`	0.1 %	0.4 %	`$comment`	0.0 %	0.0 %

(#Occ) in 289 files (#Files). Below the top summary row, we list the individual keywords, breaking down shares of matches in percent (visualized by progress bars). The right half of the table provides statistics for sub-schemas that are negated and referenced, and therefore reachable via a path `.**.not.$eref.*`.

In the following, we will omit the prefix "`.**`" from path queries, assuming the context is clear to our readers. We sorted the table on the total number of `not.`k+`not.$eref.`k occurrences (see [5] for the absolute values), and it is interesting to compare the weight of different keywords in both parts.

A `not` may not correspond to any `not.*` pattern, when followed by `{ }`. We found 16 such occurrences, expressing the schema `false`, which is not satisfied by any instance. This use of `not` is a consequence of the fact that `false` has only been introduced with Draft-06.

Complex arguments. Table 1 indicates a total of 840 occurrences of `not.`*, Figure 2 reported 787 occurrences of `not`. The values differ since the negated sub-schema can be complex. Most instances of `not` have a simple sub-schema. Most negated complex schemas have two keywords, but some have three or four.

(a)
```
"not": {
  "enum": ["markdown",
           "code",
           "raw"] }
```

(b)
```
"not": {
  "enum": ["generic-linux"],
  "type": "string" }
```

(c)
```
"not": {
  "items": {
    "not": {
      "type": "string",
      "enum": [
        "Dataset", "Image",
        "Video", "Sound",
        "Text" ] } }
```

(d)
```
{ "type" : "object",
  "oneOf": [
     { "properties":
        { "when": {"enum": ["delayed"]}},
       "required": ["when","start_in"] },
     { "properties":
        { "when": { "not": {"enum": ["delayed"]}
}}} ] }
```

(e)
```
{ "type": "object",
  "if": {
    "required": ["when"],
    "properties":
      { "when": {"enum": ["delayed"]} }},
  "then": {
    "properties":
      { "when": {"enum": ["delayed"] }},
    "required": ["when", "start_in"] }}
```

Fig. 3. JSON Schema snippets exemplifying real-world usage patterns.

The situation is very different with `$eref`, i.e., references expanded in preprocessing. Here, 93 occurrences of `not.$eref` correspond to 338 occurrences of `not.$eref.*`. Thanks to the mediation of `$eref`, the schema designer implicitly applies negation to a complex argument, with an average of 3-4 members.

Results. The most common argument of negation is `required`. The pattern `not.items` is second-most common, followed by `not.type` and `not.properties`.

While `not.required` dominates the `not.*` case, the two most common cases of the `not.$eref` group are `not.$eref.type`, whose value is `object` in 80% of the cases, and `not.$eref.properties`, which indicates that `not.$eref` is mostly used to negate complex object definitions. This explains the much higher occurrence of descriptive keywords inside the referenced argument.

4.3 RQ3: What are common real-world usage patterns?

Field and value exclusion. Field exclusion via `not.required` is the most frequent path, and this usage was already discussed in Example 1.

We discuss the paths `not.enum` and `not.const` together, as both are used to exclude values. Snippets of example schemas are shown in Figures 3(a) and (b). Such schemas have an obvious interpretation: the instance may have any type and must be different from the string or strings listed. In the majority of cases, the sub-schema is simple, as in Figure 3(a) (link to schema 89480). In the complex cases, `enum` is always paired with a `"type" : "string"` assertion, as in Figure 3(b) (link to schema 3458). This assertion is redundant, since all values listed by `enum` are strings. This co-occurrence is not specific to negation, since also in positive schemas, `enum` is paired with a `type` assertion in the vast majority of cases.

Paraphrasing `contains`. The pattern `not.items` is among the most common `not`-paths. All such schemas have either the structure `not.items.not` (as in Figure 3(c), link to schema 88916) or `not.items.enum`.

The `items` assertion is verified by any instance that is not an array, or that is an empty array, or that is an array where every element satisfies the schema associated with `items`. Hence, it is only violated by instances that are arrays, and which contain at least one element that violates the schema. While `items` specifies a universally quantified property, `not.items` can be used to specify an existentially quantified property, as does the `contains` keyword (as we will discuss shortly). The jargon `not.items.enum` specifies that the array must contain at least one value that is not listed in the argument of `enum`. The jargon `not.items.not` specifies that the instance is an array that contains at least one value that satisfies S, according to the following equivalence:

`"not"`: { `"items"`: { `"not"`: S } } $\Leftrightarrow$ {`"type"`: `"array"`, `"contains"`: S }

These two cases cover, with minimal variations, all occurrences of `not.items`. (In fact, all these schemas originate from just two groups of schema designers.)

To sum up, `not.items` can be used to express `contains`. This is an instance of a pattern that may be replaced by a single (and thus simpler) operator.

Paraphrasing Discriminated Unions. The schema snippet in Figure 3(d) (link to schema 90970) allows interesting observations about the use of `oneOf`. JSON Schema specifications do not prescribe that the branches of `oneOf` are mutually exclusive, but they state that a value must match a single branch only. However, the two branches of `oneOf` happen to be mutually exclusive: if `"when"` is absent, then only the second branch holds. If it is present, then it is associated to complementary types in the two branches, so here, `oneOf` is actually `anyOf`. Applying equivalent rewritings (from $\neg a \vee b$ to $a \Rightarrow b$, and pushing down negation), the schema can be rewritten as shown in Figure 3(e). Now the specification is more clear: if `"when"` has the value `"delayed"`, then `"start_in"` is required.

This suggests that `oneOf` is used to express a form of *discriminated unions*. In discriminated unions, also known as *tagged unions* or *labeled unions*, each branch is labeled with a unique label (or tag), and any value matching the union must be prefixed by the label of the only branch it matches.

Results. Certain usage patterns are quite common: field exclusion, value exclusion from sets of strings, and field mutual exclusion. Field exclusion is so common that one may imagine to add an ad-hoc operator to the JSON Schema language.

5 Discussion

In our analysis, we learned that negation is used in many different ways, some of which are extremely creative. In the following, we discuss our key observations.

Redundancy. Schema designers tend to overspecify by adding redundant assertions. A quantitative follow-up study, based on structured interviews, would help to understand their motivation. For instance, redundancy might be introduced to improve schema readability.

Comprehensibility. A general lesson learned is that JSON Schema semantics can be subtle, and the JSON notation can create readability problems. Educational tools for analyzing JSON Schema semantics, such as rewriting schemas to eliminate negation or even generate witnesses to schemas [2,4], may help.

Language Extensions. We observed that negation is often used in order to express, in a cumbersome way, the *discriminated unions* pattern, where the value of one field determines the presence/absence and the type of the others. This may trigger reflections about schema design.

`not.required` is heavily used to forbid the presence of properties. One may imagine adding a `"forbidden"`: [*"k1"*, ..., *"kn"*] operator as a simpler way to specify that properties *"k1"*, ..., *"kn"* cannot be present.

Benchmarks. Some of the most popular jargons found in our study are not reflected in the JSON Schema Test Suite [1], a collection of synthetic schemas for benchmarking JSON Schema validators. At the time of this writing (commit hash #09fd353 of the test suite), the test schemas only include the paths `not.type`, `not.properties`, `not` followed by `{ }`, `not.true`, as well as `not.false`. We hope that our study may help extend such test suites by popular usage patterns.

6 Related Work

We provide more details on the usage of not in our extended technical report [5].

We can safely claim that our collection of real-world JSON Schema documents is so far the most diverse: In an earlier empirical study [13], we analyzed schemas from SchemaStore,[6] a curated collection of real-world JSON Schema documents (150 at that time). This study targeted a comparatively coarse-grained classification of the occurrences of language operators, e.g., all Boolean operators were treated as a single group.

There is an established tradition of empirical studies on schema languages for semi-structured data (e.g., [6,8,12,14]). In hindsight, these studies have guided researchers towards addressing the relevant language features. Understanding how negation is used is also relevant for building practical tools, e.g., for validation [7], containment checking [2,11], or witness generation [2].

7 Summary

In our study on the usage of `not` in JSON Schema, we identified three cases: (a) We found that one reason for using `not` is that schema designers are missing certain negative dual operators, such as a `forbidden` would be the missing dual to `required`. (b) Another case that we encountered is that `not` is used to encode implication (so $a \Rightarrow b$ may be encoded as $\neg a \vee b$). (c) Finally, negation is used for subtraction, e.g., to specify integers that are not multiples of 2.

[6] SchemaStore, at https://www.schemastore.org/json/, last accessed 21-Apr-2021.

In case (c), we regard negation as useful, whereas for the other cases, we would prefer to see suitable extensions to the language.

Acknowledgments. This contribution was partly funded by *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) grant #385808805. The schemas were retrieved using Google BigQuery, supported by Google Cloud. We thank Thomas Pilz (OTH Regensburg) for his help in making the research artifacts available. We thank Michael Fruth (University of Passau) for feedback on an earlier draft.

References

1. JSON Schema Test Suite. Available at: https://github.com/json-schema-org/JSON-Schema-Test-Suite, version of commit hash #09fd353. (2021)
2. Attouche, L., Baazizi, M.A., Colazzo, D., Falleni, F., Ghelli, G., Landi, C., Sartiani, C., Scherzinger, S.: A Tool for JSON Schema Witness Generation. In: Proc. EDBT 2021. pp. 694–697 (2021)
3. Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C.: Schemas and types for JSON data: From theory to practice. In: Proc. SIGMOD 2019. pp. 2060–2063 (2019)
4. Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C., Scherzinger, S.: Not Elimination and Witness Generation for JSON Schema. In: Proc. BDA 2020 (2020)
5. Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C., Scherzinger, S.: An Empirical Study on the "Usage of Not" in Real-World JSON Schema Documents (Long Version). CoRR (2021), https://arxiv.org/abs/2107.08677
6. Bex, G.J., Neven, F., den Bussche, J.V.: DTDs versus XML Schema: A Practical Study. In: Proc. WebDB 2004 (2004)
7. Bourhis, P., Reutter, J.L., Suárez, F., Vrgoc, D.: JSON: Data model, Query languages and Schema specification. In: Proc. PODS 2017. pp. 123–135 (2017)
8. Choi, B.: What are real DTDs like? In: Proc. WebDB 2002. pp. 43–48 (2002)
9. Friesen, J.: Java XML and JSON: Document Processing for Java SE, chap. Extracting JSON Values with JsonPath, pp. 299–322. Apress (2019)
10. Fruth, M., Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C., Scherzinger, S.: Challenges in Checking JSON Schema Containment over Evolving Real-World Schemas. In: Proc. EmpER 2020. pp. 220–230 (2020)
11. Habib, A., Shinnar, A., Hirzel, M., Pradel, M.: Finding data compatibility bugs with JSON subschema checking. In: Proc. ISSTA 2021. pp. 620–632 (2021)
12. Laender, A.H., Moro, M.M., Nascimento, C., Martins, P.: An X-ray on Web-available XML Schemas. SIGMOD Rec. **38**(1), 37–42 (Jun 2009)
13. Maiwald, B., Riedle, B., Scherzinger, S.: What Are Real JSON Schemas Like? — An Empirical Analysis of Structural Properties. In: Proc. EmpER 2019. pp. 95–105 (2019)
14. Martens, W., Neven, F., Schwentick, T., Bex, G.J.: Expressiveness and Complexity of XML Schema. ACM Trans. Database Syst. **31**(3), 770–813 (Sep 2006)
15. MongoDB, Inc.: MongoDB Manual: $jsonSchema (Version 4.4) (2021), https://docs.mongodb.com/manual/reference/operator/query/jsonSchema/
16. json-schema org: JSON Schema (2021), available at https://json-schema.org
17. Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M., Vrgoč, D.: Foundations of JSON Schema. In: Proc. WWW 2016. pp. 263–273 (2016)
18. StackOverflow: JSON Schema – valid if object does *not* contain a particular property. Available at: https://stackoverflow.com/questions/30515253/json-schema-valid-if-object-does-not-contain-a-particular-property