Challenges in Checking JSON Schema Containment over Evolving Real-World Schemas

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

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

Abstract. JSON Schema is maturing into the de-facto schema language for JSON documents. When JSON Schema declarations evolve, the question arises how the new schema will deal with JSON documents that still adhere to the legacy schema. This is particularly crucial in the maintenance of software APIs. In this paper, we present the results of our empirical study of the first generation of tools for checking JSON Schema containment which we apply to a diverse collection of over 230 real-world schemas and their altogether 1k historic versions. We assess two such special-purpose tools w.r.t. their applicability to real-world schemas and identify weak spots. Based on this analysis, we enumerate specific open research challenges that are based on real-world problems.

Keywords: JSON Schema Containment · Empirical Study.

1 Introduction

With the proliferation of JSON as a data exchange format, there is a need for a schema language that describes JSON data: By relying on schema languages, software developers can reduce the burden of defensive programming, since they can trust their input to adhere to certain constraints [9]. Among various proposals for a JSON schema language (see [1] for an overview), JSON Schema (link available in the PDF) is on its way to standardization. First results on the theoretical properties of this language have already been published [4,11].

When schemas evolve as part of larger software projects, the question arises how the new schema version compares to the previous version. For instance, developers will want to know whether the new API (described by a schema) will still accept input from legacy clients; if not, developers risk runtime errors. Decisions on *JSON Schema containment*, e.g., whether the language declared by one schema is a subset of the other, require tool support. One such tool is `json-schema-diff-validator` (link available in PDF). The 1.7k–14k weekly

```
{ "properties":{        { "properties":{
     "fruit":{               "fruit":{
       "enum":[                "enum":[
           "apple",                "apple",
-          "pear"       +          "pear",
 ]  } } }               +          "banana"
                         ]  } } }
```

Fig. 1: JSON Schema document E_1 (left) is a sub-schema of E_2 (right).

downloads from `npmjs` since 5-Jan-2020 confirm a strong demand. JSON Schema containment has recently also been explored in academic research [8].

In this paper, we conduct an empirical study on tools for checking JSON Schema containment, which we refer to as *JSC-tools*: We apply JSC-tools on a diverse collection of JSON Schema documents. In particular, we set out to identify weak spots in these tools which are rooted in genuine research challenges.

Contributions. Our paper makes the following contributions:

- We apply state-of-the-art JSC-tools to schemas hosted on SchemaStore (link available in the PDF), where developers share real-world JSON Schema documents for re-use. As of today, SchemaStore is the largest collection of its kind. From the GitHub repository backing this website, we analyze over 230 schemas, with a total of over 1k historical versions.
- We investigate three research questions: (RQ1) We assess the applicability on JSC-tools on real-world schemas, i.e., the share of schemas that can be correctly processed. (RQ2) We ask which real-world language features are difficult to handle. (RQ3) We further determine the degree of consensus among JSC-tools applied to the same input, as an indicator whether classification decisions can be relied upon.
- Based on the insights thus gained, we identify open research challenges.
- We publish our fully automated analysis pipeline, to allow fellow researchers to build upon and reproduce our results.

Structure. In Section 2, we motivate that checking JSON Schema containment is not trivial. In Section 3, we describe our methodology. We address our research questions and present our results in Section 4, with a discussion of research opportunities in Section 5. We cover potential threats to validity in Section 6, and discuss related work in Section 7. Section 8 concludes.

2 Examples of JSON Schema Containment

We motivate that checking JSON Schema containment is not trivial. Our examples are based on instances of JSON Schema evolution that we have observed on SchemaStore. We basic assume familiarity with JSON syntax and otherwise refer to [4] for an introduction to the JSON data model.

Conditional semantics. Let us consider schema E_1, shown left in Figure 1. The JSON Schema language employs a conditional semantics, demanding that *if* a

```
1 { "properties":{              { "properties":{
2       "address":{                   "address":{
3 -        "type": "string"    +        "properties":{
4  }  } }                       +           "street": { "type": "string" },
5                              +           "number": { "type": "integer" },
6                              +           "city":   { "type": "string" } }
7                                }  } }
```

Fig. 2: JSON Schema document S_1 (left) is a sub-schema of S_2 (right).

JSON value is an object, and *if* that object has a property named `fruit`, then its value must be either the string `"apple"` or `"pear"`. Hence, `{"fruit": "banana"}` is invalid, yet the raw string value `"banana"` is valid, since it is not an object. Objects without a property `fruit` are also valid, such as `{"vegetable": "potato"}`.

Extending enumerations. Let us assume that the schema is changed to E_2, as shown in Figure 1 (right). We employ a `diff`-based notation, showing the original and the changed schema side-by-side. Removed lines are prefixed with minus, added lines are prefixed with plus. In line 5, a comma is added at the end of the line, and item `banana` is added in line 6. The document `{"fruit": "banana"}` is now valid w.r.t. schema E_2. We say schema E_1 is a sub-schema of E_2, since the language it defines is a subset of the language defined by E_2.

Introducing objects. Schema S_1 in Figure 2 (left) specifies that if a JSON document is an object with a property named `address`, then the value of this property is of type string. Thus, document D : `{"address": "Burbank California"}` is valid w.r.t. S_1. We now refactor the schema to S_2, as shown. Document D is still valid w.r.t. the new schema S_2, as the conditional semantics only imposes restrictions if the type of the address is an object. On the other hand, a JSON document with an address structured as an object is not valid w.r.t. schema S_1, which expects a string. Hence, S_1 is a sub-schema of S_2.

Adding new properties. We continue with schema S_2 and extend the address properties by `"zip": {"type": "integer"}` (inserted after line 6 in Figure 2 on the right), declaring that ZIP codes must be integer values. We refer to this new schema as S_3. Schema S_2 allows any type for the ZIP code (e.g. string: `"zip": "1234"`), as additional properties are allowed by default. Thus, schema S_3 is more restrictive than S_2 and therefore a sub-schema of S_2.

Summary. Reasoning whether schema containment holds is not trivial, even for toy examples. With real-world schemas, which can be large and complex [10], we absolutely need the support of well-principled tools. Assessing the state-of-the-art in such JSC-tools is the aim of our upcoming empirical study.

3 Methodology

3.1 Context Description

Schema collection. We target the JSON Schema documents hosted on SchemaStore, a website backed by GitHub, as of 19-Jun-2020 (commit hash c48c727).

JSC-Tools. The JSC-tool `json-schema-diff-validator`, mentioned in the Introduction, only compares syntactic changes: nodes added, removed, and replaced are considered breaking changes. This can lead to incorrect decisions regarding schema containment, e.g., for the schemas from Figure 2. We therefore exclude this tool from our analysis.

Instead, we consider two tools that perform a *semantic* analysis, one tool from academia, and another from an open source development project. Since the tools have rather similar names, we refer to them as Tool A and Tool B:

- Tool A is called `jsonsubschema` (link available in the PDF) and is an academic prototype implemented in Python, based on well-principled theory [8]. Based on the authors' recommendation (in personal communication), we use the GitHub version with commit hash 165f893.

  Tool A supports JSON Schema Draft 4 without recursion, and has only limited support for negation (`not`) and union (`anyOf`).
- Tool B, `is-json-schema-subset` (link available in the PDF) is also open source and implemented in TypeScript. We use the most recent version available at the time of our analysis (version 1.1.15). Tool B supports JSON Schema Drafts 5 and higher. No further limitations are stated.

3.2 Analysis Process

Our data analysis pipeline is fully automated. The Python 3.7 scripts for our data preparation and analysis pipeline, as well as the raw input data, are available for reproduction analysis.[6] We use the Python modules `jsonschema` (version 3.2.0) and `jsonref` (version 0.2) for JSON Schema validation and dereferencing.

Obtaining schema versions. We retrieve the historic versions of all JSON Schema documents hosted on SchemaStore from the master branch of its GitHub repository (link available in the PDF), provided that they are reachable by path `src/schemas/json/`. This yields 248 schemas. About half of them have not changed since their initial commit, while some schemas count over 60 historic versions. In total, we obtain 1,069 historic schema versions which we have validated to ensure they are syntactically correct JSON Schema documents.

Excluding schema versions from analysis. One practical challenge is that the JSC-tools considered support non-overlapping drafts of JSON Schema, while we need to process the same document with both tools. As a workaround, we determine the subset of documents that are both valid w.r.t. Draft 4, Draft 6, and Draft 7, thereby excluding four documents.

[6] https://github.com/michaelfruth/jsc-study

Table 1: Comparing both JSC-tools in two separate experiments: (a) Table 1a shows *reflexivity* of schema equivalence ($\equiv$) for all 1,028 schemas ($\perp$ denotes runtime errors). Table rows show results for Tool A, columns for Tool B. (b) Table 1b states results of checking 796 pairs of *successive schema versions* w.r.t. equivalence, strict containment ($\subset, \supset$), incomparability ($||$), and runtime errors.

(a) Schema reflexivity.

Tool A	Tool B		
	$\equiv$	$\perp$	Σ
$\equiv$	36.9%	0.2%	37.1%
$\perp$	48.2%	14.7%	62.9%
Σ	85.1%	14.9%	100.0%

(b) Succeeding schema versions.

Tool A	Tool B				
	$\equiv$	$\subset$	$\supset$	$\Vert$	$\perp$
$\equiv$	**9.5%**	0.3%	0.3%	0.4%	0.0%
$\subset$	3.0%	**2.9%**	0.3%	1.6%	0.0%
$\supset$	5.7%	0.0%	**1.0%**	1.1%	0.0%
$\Vert$	3.4%	0.6%	0.5%	**2.6%**	0.4%
$\perp$	25.9%	3.3%	0.5%	17.1%	**19.7%**

Regarding drafts, we need to take further care: JSON Schema is designed as an open standard, which means that a validator will accept/ignore unknown language elements that are introduced in a future draft. Then, running both tools on the same schema document constitutes an unfair comparison, since the tools will have to treat these elements differently. We therefore search for keywords introduced/changed *after* Draft 4 (e.g., `const` or `if-then-else`). In total, we thus exclude 41 documents, a choice that we also discuss in Section 6.

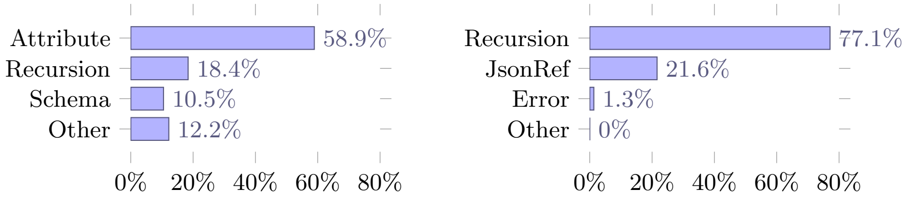
Overall, we obtain 1,028 JSON Schema documents, where approx. 10% contain recursive references. We count 232 schemas in their latest version and 796 pairs of documents that are two versions of the same schema, ordered by the time of their commits, where no other commit has changed the schema in between. In the following, we refer to such pairs as *successive schema versions*.

4 Detailed Study Results

4.1 RQ1: What is the real-world applicability of JSC-tools?

We are interested in the share of real-world schemas that the JSC-tools can reliably process. This is an indicator whether these tools are operational in practice. To this end, we perform a basic check: Given a valid JSON Schema document S, equivalence is reflexive ($S \equiv S$). Given this ground truth, we compare each schema version with itself.

The results are shown in Table 1a. The first row states the percentage of documents that Tool A recognizes as equivalent. The second row states the percentage of documents where Tool A fails (denoted "$\perp$"). "Σ" shows sums over rows/columns. The results for Tool B are shown in columns. The top left entry states that for less than half of the documents, both tools agree they are equivalent to itself. About 15% of documents cannot be checked by either tool.



(a) Tool A.

(b) Tool B.

Fig. 3: Error distribution (in %) for the experiment from Table 1a.

Results. We observe a high failure rate for Tool A. In the experiments conducted by Habib et al. [8], the authors of Tool A, Tool B performs comparatively worse than Tool A. Further investigations, performing experiments with the exact same version of Tool B as used in [8], have revealed that the applicability of Tool B has meanwhile improved. Moreover the experiments in [8] consider a different schema collection, as we also discuss in Section 7.

Since not all real-world language features are supported by Tool A, our first experiment is evidently setting up Tool A for failure. We next look more closely into which language features are problematic.

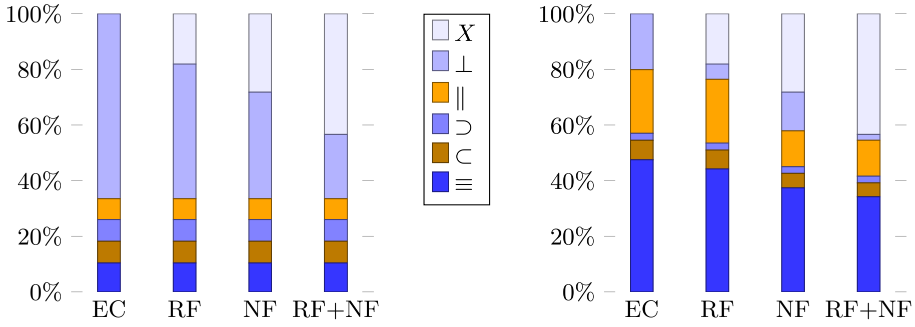
4.2 RQ2: Which language features are difficult to handle?

We are interested in which properties of real-world schemas cause JSC-tools to fail, either because not yet supported or incorrectly handled. As a first step, we inspect the error messages for documents that cannot be processed in the first experiment. Figure 3 visualizes the distribution of the top-3 runtime errors. While the tools use different names in error reporting, it is obvious that recursion and reference errors are frequent.

To further investigate which operators of JSON Schema are problematic, we consider subsets of our document collection, where we exclude schemas with certain language features. In particular, we check pairs of successive schema versions for containment. We register when a tool decides that the schema versions are equivalent; if not, whether the language declared by the predecessor version is a sub-set of the language declared by the successor version, or a super-set. In all remaining non-error cases, we consider the versions incomparable.

In Figure 4, we show the relative results for (i) the entire collection, (ii) a subset where all references are non-recursive, contain only document-internal references or references to URLs, which can be resolved, (iii) a subset without `not`, and (iv) the combination of all these restrictions. For Tool A, the classification decisions remain identical throughout, only the error rate decreases. This means we have indeed excluded the problematic schema documents. With Tool B, the classification decisions vary slightly, but we see the error rate decrease to ca. 2%.

Results. Recursion and negation are obvious challenges for JSC-tools. While Tool A explicitly does not support recursion, and negation only to some extent,



(a) Tool A.

(b) Tool B.

Fig. 4: Checking pairs of successive schemas on (i) the entire collection (EC: 796 pairs), (ii) a subset where all references are non-recursive, document-internal or URLs, and can be resolved (RF: 652 pairs), (iii) without `not` (NF: 572 pairs), and (iv) the combination of (ii) and (iii) (RF+NF: 451 pairs). "X" represents excluded schema documents. Reporting decisions in % of the entire collection.

Tool B (where no limitations are specified) struggles with these language constructs as well. However, recursion and negation do occur in real-world JSON Schema documents, and we refer to Section 5 for a discussion of which use cases for JSON Schema are affected when these features are not supported.

4.3 RQ3: What is the degree of consensus among JSC-tools?

To assess how well both tools agree, we compare successive schema versions w.r.t. the classification decisions of both tools. Table 1b summarizes the results. Again, results for Tool A are shown in rows, results for Tool B in columns. In an ideal world, the JSC-tools completely agree, so we expect a diagonal matrix (with zeroes in all cells except on the diagonal). However, the tools disagree *considerably*. For instance, for 5.7% of the pairs, Tool A claims that the first schema version declares a super-language of the second, while Tool B regards both versions as equivalent (row ⊃/column ≡). The tools agree on only approx. 50% of subset of inputs that both tools can process without a runtime error.

Results. Evidently, the degree of consensus is low. Since developers cannot yet rely on JSC-tools, they are forced to visually compare evolving schemas, near-impossible for the complex and large schemas encountered in the real world (some schemas on SchemaStore take up over 10MB stored on disk [10]). In the upcoming discussion, we discuss open research questions in this context.

5 Discussion of Results and Research Opportunities

Summary. Our experiments show that the first generation of JSC-tools is still in an early stage where recursion and negation in schemas are not yet well covered.

In earlier work [10], we have manually categorized all SchemaStore schemas depending on their purpose: *data* schemas use JSON primarily as a data format. *meta* schemas define markup for other schemas. For instance, there are JSON *meta* schemas for every JSON Schema Draft. *conf* schemas describe JSON documents that configure services. *app* schemas are used for data exchange between applications.[7] This categorization provides a general overview how JSON Schema is employed in practice. Aligning the documents excluded in the experiments from Figure 4 reveals that by ignoring recursive schemas, we primarily exclude *conf* and *meta* schemas. By ignoring schemas with negation, we again mainly exclude *conf* schemas. In summary, the JSC-tools best cover *data* and *app* schemas, while on SchemaStore, *conf* schemas constitute the largest group.

Research Opportunities. Making JSC-tools operable for production is more than just an engineering effort, and we see several opportunities for impactful research:

- Handling *recursion and negation* in checking JSON Schema containment is still unresolved. As recursion combined with negation is a general challenge in database theory, e.g., when specifying sound semantics for Datalog, we may expect some concepts to transfer (as also proposed in [4]).
- Not only do practitioners need robust and complete tools for inclusion checking, they also need to understand why containment holds/does not hold:
  - This could be done by means of *instance generation*, i.e., generating small example schemas that capture why the schemas differ. A first proposal for witness generation is sketched in [2].
  - Alternatively, pointing to the positions in the schema declarations that cause containment checks to fail would also provide some degree of *explainability* for developers comparing schemas.
- Having reliable JSC-tools at hand would allow us to build editors that assist with schema refactoring. Then, a JSC-tool can confirm that the refactored schema is still equivalent to its original (while the new version may be easier to comprehend, easier to validate, or simply more succinct).
- We have come to realize that the data management community is in need of a dedicated micro-benchmark for JSC-tools, with small yet realistic documents where we know the ground truth w.r.t. schema containment. Such a benchmark would certainly benefit researchers and practitioners alike.

6 Potential Threats to Validity

Supported JSON Schema drafts. As stated in Section 3, the JCS-tools employed in this study support different drafts of JSON Schema: Tool A handles Draft 4, Tool B handles Draft 5+. Due to the open standard policy of JSON Schema, a keyword introduced in Draft 5+ is ignored by Tool A, but will carry meaning for Tool B. As discussed, in order to level the playing field, we exclude affected documents from our analysis: We have diligently checked for keywords, and regard the risk that we may have overlooked problematic documents as minor.

[7] Naturally, such a classification is always subjective, and not necessarily unique, as also remarked in the original work on DTDs [5] that inspired this categorization.

Recursion in schemas. We noticed minor differences in reporting recursion errors in the Python-based Tool A and the TypeScript-based Tool B, which we trace back to the different programming languages and their libraries. Specifically, we noticed spurious, non-deterministic behavior in the `jsonschema` module for recursive schemas. However, this affects only a handful of JSON Schema documents and is a minor threat to our results at large.

Renaming schema files. In our analysis, we consider all historic versions of a schema based on `git` commits. For `git` to recognize that a file is renamed, the content of the file must remain the same. Due to (so far) 13 renamings in the history of SchemaStore, our collection contains 13 duplicate schema versions. Since this is a small number in the context of over 1k schema versions analyzed, this again is a minor threat.

Mining `git` *and GitHub.* In mining `git` repositories, we face the usual and well understood threats [3]. Our analysis is based on a specific schema collection. One threat to validity is that the historic schema versions in this collection are skewed: Only eight schemas account for almost one third of all historic schema versions. Nevertheless, SchemaStore is to date the largest and also most diverse collection of JSON Schema documents, and thus highly suitable for our purposes.

7 Related Work

There is a mature body of work on schema containment for XML schemas (such as DTDs and XML Schema), e.g. [9], based on automata as the formal vehicle [7]. Other approaches exist that rather relay on a particular class of constraints to check inclusion between XML schemas like Extended DTDs featuring regular expressions with interleaving and counting [6].

First theoretical properties of the JSON Schema language have been studied recently [4,11]. To the best of our knowledge, the tool by Habib et al. [8] ("Tool A" in our study) is the first academic exploration of JSON Schema containment. Their experiments are closest to our work, since they also compare Tools A and B on real-world JSON Schema documents, but they use a different baseline. In particular, the authors choose three sources for JSON Schema documents, while SchemaStore hosts schemas from over 200 different sources.

It seems plausible that the schema collection studied by us is not only larger in terms of the number of distinct schemas (each with its historic versions), but overall also more diverse, while skewed towards schemas for configuring services [10]. This may explain some of the differences in our respective experiments regarding the successful applicability to real-world schemas, plus the fact that Tool B has meanwhile been improved (see our discussion in Section 4.1).

In general, checking JSON Schema containment is not a trivial task: As JSON Schema is not an algebraic language, syntactic and semantic interactions between different keywords in the same schema object complicate programmatic handling. In [2], we therefore propose a dedicated algebra for JSON Schema, which has the potential to serve as a formal foundation for new approaches to checking JSON Schema containment.

8 Conclusion

In this paper, we evaluate a first generation of tools for checking JSON Schema containment. Our analysis shows that this is still a *very* young field, with open research opportunities that have immediate practical relevance.

In particular, we recognize the need for a micro-benchmark for JSC-tools. While there is a well-adopted benchmark for JSON Schema validation, the JSON Schema Test Suite (link available in the PDF), no comparable benchmark exists for checking JSON Schema containment, with pairs of documents for which containment was determined manually, for a particular operator or a logical group of operators. Such a micro-benchmark could be inspired by real-world schemas found on SchemaStore. We plan to address this in our future work.

Acknowledgments: This project was partly supported by the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation), grant #385808805.

References

1. Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C.: Schemas and Types for JSON Data: From Theory to Practice. In: Proceedings of the 2019 International Conference on Management of Data (SIGMOD). pp. 2060–2063 (2019)
2. Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C., Scherzinger, S.: Not Elimination and Witness Generation for JSON Schema. In: BDA '20 (2020)
3. Bird, C., Rigby, P.C., Barr, E.T., Hamilton, D.J., Germán, D.M., Devanbu, P.T.: The promises and perils of mining git. In: Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR). pp. 1–10 (2009)
4. Bourhis, P., Reutter, J.L., Suárez, F., Vrgoc, D.: JSON: Data model, Query languages and Schema specification. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS). pp. 123–135 (2017)
5. Choi, B.: What are real DTDs like? In: Proceedings of the Fifth International Workshop on the Web and Databases (WebDB). pp. 43–48 (2002)
6. Colazzo, D., Ghelli, G., Pardini, L., Sartiani, C.: Efficient asymmetric inclusion of regular expressions with interleaving and counting for XML type-checking. Theor. Comput. Sci. **492**, 88–116 (2013)
7. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications. Available on: http://tata.gforge.inria.fr/ (2007), release October, 12th 2007
8. Habib, A., Shinnar, A., Hirzel, M., Pradel, M.: Type Safety with JSON Subschema. CoRR **abs/1911.12651v2** (2020), http://arxiv.org/abs/1911.12651v2
9. Lee, T.Y.t., Cheung, D.W.l.: XML Schema Computations: Schema Compatibility Testing and Subschema Extraction. In: Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM). p. 839–848 (2010)
10. Maiwald, B., Riedle, B., Scherzinger, S.: What Are Real JSON Schemas Like? - An Empirical Analysis of Structural Properties. In: Advances in Conceptual Modeling - ER 2019 Workshop EmpER. pp. 95–105 (2019)
11. Pezoa, F., Reutter, J.L., Suárez, F., Ugarte, M., Vrgoc, D.: Foundations of JSON Schema. In: Proceedings of the 25th International Conference on World Wide Web (WWW). pp. 263–273 (2016)