Witness Generation for JSON Schema Patterns

Christoph Köhnen[1]

Abstract: JSON Schema is a schema language for the popular data exchange format JSON. This paper introduces an approach to convert regular expressions, which appear in `ECMA-262` syntax in JSON Schema, into an alternative syntax, such that they may be compiled to finite-state automata. Specifically, we address the challenge that the `ECMA-262` pattern syntax uses anchor symbols to mark the beginning and end of a word, which is not compatible with available libraries for automata manipulation. This is a step towards generating witnesses, i.e., JSON instances which are valid w.r.t. the given JSON Schema specification. We implement an algorithm proposed by Dominik Freydenberger to convert regular expressions into `brics` syntax. We show that we successfully address over 97% of the unique patterns found in a collection of thousands of JSON Schema specifications collected from GitHub.

1 Introduction

JSON Schema is a language for describing collections of JSON instances, where JSON is a widely adopted format for data exchange. This article describes a *student Bachelor thesis* project which targets a key problem in generating a *witness* for a given JSON Schema specification (short *schema*), which is a JSON instance valid w.r.t. this given schema. Witness generation has several important applications, such as checking schema containment [At22], a challenge also researched in [Ha21] for type-checking data science pipelines.

For example, consider the JSON Schema specification in Figure 1. Any witness must be of type object, as required in line 1. If the object has a property (line 2) whose name matches the pattern `^(sur)?name$` (line 3), so `"surname"` or `"name"`, then its value must be a string (line 4) matching the regular expression in line 5 with at least three characters (line 6). As this example illustrates, JSON Schema *patterns* can describe property keys (via `patternProperties`) or string-typed values (via the keyword `pattern`).

JSON Schema patterns are encoded in ECMA-262 syntax[2] and are actively used in practice: In analyzing a corpus of approx. 80K open source JSON Schema documents [Ba21], we found that 21% of the schemas contain patterns. Our approach cannot handle lookahead and lookbehind as well as the word boundaries `\b` and `\B`. The former one matches between a word and a non-word character without consuming one. But this concerns less than 3% of the unique patterns found in this corpus. We further found that the bulk of patterns actually describes regular languages. Some patterns in `ECMA-262` can exceed the expressiveness of

[1] Universität Passau, Fakultät für Informatik und Mathematik, Lehrstuhl für Informatik mit Schwerpunkt Skalierbare Datenbanksysteme, Innstraße 33, 94032 Passau, Deutschland, koehne02@ads.uni-passau.de
[2] https://json-schema.org/draft/2020-12/json-schema-core.html#rfc.section.6.4

doi:10.18420/BTW2023-79

regular languages, due to backreferences [FS19]. We consider such occurrences as normal characters.

Translating the remaining patterns into finite-state-automata [HU79], we can generate string witnesses for patterns, simply by traversing a path from the initial state to some accepting state. From product automata, we can generate string witnesses that adhere to several constraints, e.g., matching *several* patterns as well as minimum and maximum lengths.

Our Java implementation of a tool for JSON Schema witness generation [At22] uses the automaton library `brics` [Mø17] for creating automata from regular expressions, as well as for computing automaton operations. However, `brics` relies on the syntax for regular expressions accustomed from computer science textbooks [HU79], assuming expressions to be *bounded*. Thus, the expression `bc*` matches `"b"` and `"bc"`, but not `"abcd"`. Yet in `ECMA-262` syntax, the equivalent regular expression would have to be explicitly bounded as `^bc*$`.

```
1 { "type": "object",
2   "patternProperties": {
3     "^(sur)?name$": {
4       "type": "string",
5       "pattern": "^[A-Z][a-z]*$",
6       "minLength": 3 }}}
```

Fig. 1: A JSON Schema specification.

Contributions. (1) We implement a novel algorithm to convert patterns from `ECMA-262` to `brics` syntax. The conversion was suggested to us by Dominik Freydenberger. (2) We have integrated our implementation in a tool for JSON Schema witness generation [At22]. (3) We further present an empirical study on the applicability of our approach to patterns as they appear in tens of thousands of real-world schemas crawled from GitHub and provide a fully automated reproduction package of it. We can show that for 97% of the unique patterns found in this corpus, we can successfully apply our rewriting.

Structure. Section 2 introduces preliminaries and presents the conversion of regular expressions in `ECMA-262` syntax to `brics` syntax. Section 3 presents and discusses our empirical evaluation. Section 4 reviews related work. Section 5 concludes.

2 Patterns and Witness Generation

$$
\begin{aligned}
\mathit{expression} &::= \mathit{basicValue} \mid \mathit{object} \mid \mathit{array}\ ; \\
\mathit{basicValue} &::= \text{null} \mid \text{true} \mid \text{false} \mid x \mid s\ ; \\
\mathit{object} &::= \{k_1 : J_1, \ldots, k_n : J_n\}\ ; \\
\mathit{array} &::= [J_1, \ldots, J_n]\ ;
\end{aligned}
$$

Fig. 2: JSON grammar, adapted from [At22].

Preliminaries. The JavaScript Object Notation (JSON) is a data format where a *JSON document* (or *JSON expression*) has a syntax which is defined by the grammar in Figure 2, where $n \geq 0$, x is a number, s is a string, $J_1, \ldots, J_n$ are JSON expressions, and all k_i are pairwise different key strings. JSON Schema also uses JSON syntax. A formal semantics is defined in [At22; Pe16]. Specifically, we are interested in the keywords `patternProperties` and `pattern`, as illustrated in Figure 1.

$$
\begin{aligned}
expr &::= (expr\ \texttt{alt})?\ seq\ ;\\
seq &::= (char \mid group \mid qExpr)^*\ ;\\
qExpr &::= (char \mid group)\ quant\ ;
\end{aligned}
$$

Fig. 3: JSON Schema pattern grammar.

Approach. We denote a regular expression extracted from a JSON Schema document as *JSON Schema pattern* (or short *pattern*). Since such a pattern follows the `ECMA-262` syntax it can be defined by the grammar in Figure 3 with the following tokens: `alt` is the alternation symbol `|`, *group* one of the alternatives (*expr*), `(?:`*expr*), `(?<`*str*`>`*expr*), `(?!`*expr*), `(?=`*expr*), `(?<!`*expr*), `(?<=`*expr*), *char* a character (or its unicode representation), a character class (like `[abc]`, `[a-z]`, `[abcA-Z]` or `[^a-z]`) or an anchor symbol (`^` or `$`), *str* a sequence of characters and *quant* a simple (`+`, `*` or `?`) or range quantifier (`{m}`, `{m,n}` or `{m,}`, $n \geq m \geq 0$).

The speciality of the `ECMA-262` language is the use of the anchor symbols `^` for the beginning of a word and `$` for the end. The expression `^abc$` in `ECMA-262` syntax matches the string `"abc"` and nothing else while `abc` matches any string with `"abc"` inside, for example `"012abcdef`. To generate a string which matches a given regular expression in `ECMA-262` syntax it can be helpful to create a finite-state automaton, since every regular language can be defined by such an automaton. The Java library `dk.brics` [Mø17] supports the creation of finite-state automata with regular expressions as well as the common operations of automata like concatenation, union, intersection or negation. Matching operations are also supported. Regular expressions are defined in the class `dk.brics.automaton.RegExp`.

A *regular expression in* `brics` *syntax* consists of the tokens listed in Figure 3, the symbols `@` for any string and `#` for the empty language, but without anchor symbols, non- and named-capturing grouping and lookahead/-behind. It follows the same grammar with the exception that `grouping` only stands for simple grouping. Lookahead and lookbehind exceed the power of regular expressions. Freydenberger and Schmid worked that out in [FS19].

We now introduce an algorithm[3] to convert patterns from `ECMA-262` to `brics` syntax, i.e. removes the anchor symbols, which works for nearly all patterns found in the corpus of JSON files from open-source projects on GitHub [Ba21]. To get rid of `^` resp. `$` we define the functions `h` and `nh` (short for `hat` and `nohat`) resp. `d` and `nd` (short for `dollar` and `nodollar`). `h` and `nh` compute regular expressions which do not use a symbol for the beginning of the word, `h`(α) matches the same language as the part of α, where `^` is used, `nh`(α) the same language as the part of α, where `^` is not used. `d` and `nd` compute regular expressions which do not use symbols for the beginning or end of the word, `d`(β) matches the same language as the part of β, where `$` is used, `nd`(β) the same language as the part of β, where `$` is not used. First we define a special predicate.

Definition 2.1 *Let α be a regular expression in* `ECMA-262` *syntax. The predicate* `nullable`(α) *is* `true` *if and only if the empty word is contained in the language defined by α.*

Examples for nullable patterns are `(^a$)*`, `(a|b?)` or `^$` which only accepts the empty word. Now we can define functions to compute the anchor and non-anchor parts of a pattern.

[3] Idea, algorithm, and replacement rules by Dr. Dominik D. Freydenberger (not published).

Tab. 1: Rules to compute the values $\mathtt{h}(\alpha)$, $\mathtt{nh}(\alpha)$, $\mathtt{d}(\alpha)$ and $\mathtt{nd}(\alpha)$ for a regular expression α recursively.

α	$\mathtt{h}(\alpha)$	$\mathtt{nh}(\alpha)$	$\mathtt{d}(\alpha)$	$\mathtt{nd}(\alpha)$
$\alpha \in \{\mathtt{()}, \mathtt{@}, \mathtt{\#}\} \cup \mathtt{CHAR}$	$\mathtt{\#}$	α	$\mathtt{\#}$	α
$\mathtt{\hat{}}$	$\mathtt{()}$	$\mathtt{\#}$	$\mathtt{\#}$	$\mathtt{\hat{}}$
$\mathtt{\$}$	$\mathtt{\#}$	$\mathtt{\$}$	$\mathtt{()}$	$\mathtt{\#}$
$\alpha_1 \mid \alpha_2$	$\mathtt{h}(\alpha_1) \mid \mathtt{h}(\alpha_2)$	$\mathtt{nh}(\alpha_1) \mid \mathtt{nh}(\alpha_2)$	$\mathtt{d}(\alpha_1) \mid \mathtt{d}(\alpha_2)$	$\mathtt{nd}(\alpha_1) \mid \mathtt{nd}(\alpha_2)$
$\tilde{\alpha}^+$	$\mathtt{h}(\tilde{\alpha}) \cdot \mathtt{nh}(\tilde{\alpha})^*$	$\mathtt{nh}(\tilde{\alpha})^+$	$\mathtt{nd}(\tilde{\alpha})^* \cdot \mathtt{d}(\tilde{\alpha})$	$\mathtt{nd}(\tilde{\alpha})^+$
$\tilde{\alpha}^*$	$\mathtt{h}(\tilde{\alpha}) \cdot \mathtt{nh}(\tilde{\alpha})^*$	$\mathtt{nh}(\tilde{\alpha})^*$	$\mathtt{nd}(\tilde{\alpha})^* \cdot \mathtt{d}(\tilde{\alpha})$	$\mathtt{nd}(\tilde{\alpha})^*$
$\tilde{\alpha}?$	$\mathtt{h}(\tilde{\alpha})$	$\mathtt{nh}(\tilde{\alpha})?$	$\mathtt{d}(\tilde{\alpha})$	$\mathtt{nd}(\tilde{\alpha})?$
$\alpha_1\alpha_2$	$\mathtt{h}(\alpha_1) \cdot \mathtt{nh}(\alpha_2) \mid \mathtt{h}^{\mathtt{n}}_{\alpha_1}(\alpha_2)$	$\mathtt{nh}(\alpha_1) \cdot \mathtt{nh}(\alpha_2)$	$\mathtt{d}^{\mathtt{n}}_{\alpha_2}(\alpha_1) \mid \mathtt{nd}(\alpha_1) \cdot \mathtt{d}(\alpha_2)$	$\mathtt{nd}(\alpha_1) \cdot \mathtt{nd}(\alpha_2)$
$\tilde{\alpha}^{\{m\}}$	$\mathtt{h}(\tilde{\alpha}^{\{m,m\}})$	$\mathtt{nh}(\tilde{\alpha}^{\{m,m\}})$	$\mathtt{d}(\tilde{\alpha}^{\{m,m\}})$	$\mathtt{nd}(\tilde{\alpha}^{\{m,m\}})$
$\tilde{\alpha}^{\{m,n\}}$ $(m < 2, n > 0)$	$\mathtt{h}(\tilde{\alpha}) \cdot \mathtt{nh}(\tilde{\alpha})^{\{0,n-1\}}$	$\mathtt{nh}(\tilde{\alpha})^{\{m,n\}}$	$\mathtt{nd}(\tilde{\alpha})^{\{0,n-1\}} \cdot \mathtt{d}(\tilde{\alpha})$	$\mathtt{nd}(\tilde{\alpha})^{\{m,n\}}$
$\tilde{\alpha}^{\{m,\}}$ $(m < 2)$	$\mathtt{h}(\tilde{\alpha}) \cdot \mathtt{nh}(\tilde{\alpha})^*$	$\mathtt{nh}(\tilde{\alpha})^{\{m,\}}$	$\mathtt{nd}(\tilde{\alpha})^* \cdot \mathtt{d}(\tilde{\alpha})$	$\mathtt{nd}(\tilde{\alpha})^{\{m,\}}$
$\tilde{\alpha}^{\{m,n\}}$ $(m \geq 2, n > 0)$	$\mathtt{h}(\tilde{\alpha} \cdot \tilde{\alpha} \cdot \tilde{\alpha}^{\{m-2,n-2\}})$	$\mathtt{nh}(\tilde{\alpha})^{\{m,n\}}$	$\mathtt{d}(\tilde{\alpha}^{\{m-2,n-2\}} \cdot \tilde{\alpha} \cdot \tilde{\alpha})$	$\mathtt{nd}(\tilde{\alpha})^{\{m,n\}}$
$\tilde{\alpha}^{\{m,\}}$ $(m \geq 2)$	$\mathtt{h}(\tilde{\alpha} \cdot \tilde{\alpha} \cdot \tilde{\alpha}^{\{m-2,\}})$	$\mathtt{nh}(\tilde{\alpha})^{\{m,\}}$	$\mathtt{d}(\tilde{\alpha}^{\{m-2,\}} \cdot \tilde{\alpha} \cdot \tilde{\alpha})$	$\mathtt{nd}(\tilde{\alpha})^{\{m,\}}$

Definition 2.2 *Denote the set of all regular expressions by* REGEXP *and the set of all non-anchor character symbols and classes by* CHAR*. Let α be a regular expression in* ECMA-262 *syntax. Define the functions* $\mathtt{h}, \mathtt{nh} : \mathtt{REGEXP} \to \mathtt{REGEXP}$ *as follows. Let $\tilde{\alpha}, \alpha_1, \alpha_2$ be regular expressions in* ECMA-262 *syntax and $\mathtt{h}^{\mathtt{n}}_{\alpha_1}(\alpha_2)$ be $\mathtt{h}(\alpha_2)$ if $\mathtt{nullable}(\alpha_1)$ and $\mathtt{\#}$ otherwise. Compute $\mathtt{h}(\alpha)$ and $\mathtt{nh}(\alpha)$ by applying recursively the rules from Table 1. Analoguously, for a regular expression β in* ECMA-262 *syntax which does not use a symbol for the beginning of the word define the functions* $\mathtt{d}, \mathtt{nd} : \mathtt{REGEXP} \to \mathtt{REGEXP}$ *as follows. Let $\tilde{\beta}, \beta_1, \beta_2$ be regular expressions in* ECMA-262 *syntax which does not use a symbol for the beginning of the word and $\mathtt{d}^{\mathtt{n}}_{\beta_2}(\beta_1)$ be $\mathtt{d}(\beta_1)$ if $\mathtt{nullable}(\beta_2)$ and $\mathtt{\#}$ otherwise. Compute $\mathtt{d}(\beta)$ and $\mathtt{nd}(\beta)$ by applying recursively the rules from Table 1.*

The rules in Table 1 are complete for all possible patterns in ECMA-262 syntax not containing word boundaries, lookahead or lookbehind since these features can match inside a pattern without consuming a character. Finally, we can formulate the algorithm.

Algorithm 1 Algorithm to convert a regular expression from ECMA-262 to brics syntax.

Input: regular expression α in ECMA-262 syntax without word boundaries, lookahead or lookbehind

1: $\beta \leftarrow \mathtt{h}(\alpha) \mid \mathtt{@} \cdot \mathtt{nh}(\alpha)$ with $\mathtt{h}(\alpha)$ and $\mathtt{nh}(\alpha)$ computed using Definition 2.2.
2: $\gamma \leftarrow \mathtt{d}(\beta) \mid \mathtt{nd}(\beta) \cdot \mathtt{@}$ with $\mathtt{d}(\beta)$ and $\mathtt{nd}(\beta)$ computed using Definition 2.2.

Output: regular expression γ in brics syntax

Example 2.3 *For $\alpha = (\mathtt{\hat{}a\$} \mid \mathtt{b})?$ in* ECMA-262 *syntax we apply Algorithm 1. First, we remove $\mathtt{\hat{}}$ by applying the rules from Table 1. We obtain*

$$\mathtt{h}(\alpha) = \mathtt{h}((\mathtt{\hat{}a\$} \mid \mathtt{b})?) = \mathtt{h}(\mathtt{\hat{}a\$} \mid \mathtt{b}) = \mathtt{h}(\mathtt{\hat{}a\$}) \mid \underbrace{\mathtt{h}(\mathtt{b})}_{=\mathtt{\#}} = \underbrace{\mathtt{h}(\mathtt{\hat{}})}_{=\mathtt{()}} \cdot \underbrace{\mathtt{nh}(\mathtt{a\$})}_{=\mathtt{a\$}} \mid \underbrace{\mathtt{h}(\mathtt{a\$})}_{=\mathtt{\#}} = \mathtt{a\$},$$

$$\mathtt{nh}(\alpha) = \mathtt{nh}((\mathtt{\hat{}a\$} \mid \mathtt{b})?) = \mathtt{nh}(\mathtt{\hat{}a\$} \mid \mathtt{b})? = (\mathtt{nh}(\mathtt{\hat{}a\$}) \mid \mathtt{nh}(\mathtt{b}))? = (\mathtt{\#} \mid \mathtt{b})? = \mathtt{b}?$$

since $\mathtt{nh(\hat{}a\$)} = \mathtt{nh(\hat{})} \cdot \mathtt{nh(a\$)} = \mathtt{\#} \cdot \mathtt{nh(a\$)} = \mathtt{\#}$, $\mathtt{\hat{}a}$ *is nullable and by using the shortcuts* $\mathtt{h}(\tilde{\alpha}) = \mathtt{\#}$ *if* $\tilde{\alpha}$ *does not contain a* $\mathtt{\hat{}}$, $\mathtt{\#} \mid \tilde{\alpha} = \tilde{\alpha} \mid \mathtt{\#} = \tilde{\alpha}$ *and* $\mathtt{()} \cdot \tilde{\alpha} = \tilde{\alpha}$. *We get* $\beta = \mathtt{a\$} \mid \mathtt{@} \cdot \mathtt{b?}$. *Then we remove* $ *analoguously by applying the rules from Table 1 and obtain*

$$\mathtt{d}(\beta) = \mathtt{d(a\$)} \mid \mathtt{d(@} \cdot \mathtt{b?)} = \mathtt{d(a\$)} \mid \mathtt{\#} = \mathtt{d(a)} \mid \mathtt{nd(a)} \cdot \mathtt{d(\$)} = \mathtt{\#} \mid \mathtt{a} \cdot \mathtt{()} = \mathtt{a},$$
$$\mathtt{nd}(\beta) = \mathtt{nd(a)} \cdot \mathtt{nd(\$)} \mid \mathtt{nd(@)} \cdot \mathtt{nd(b)?} = \mathtt{a} \cdot \mathtt{\#} \mid \mathtt{@} \cdot \mathtt{b?} = \mathtt{\#} \mid \mathtt{@} \cdot \mathtt{b?} = \mathtt{@} \cdot \mathtt{b?}$$

since $ *is nullable and by using the shortcuts* $\mathtt{d}(\tilde{\beta}) = \mathtt{\#}$ *if* $\tilde{\beta}$ *does not contain a* $, $\mathtt{\#} \mid \tilde{\beta} = \tilde{\beta} \mid \mathtt{\#} = \tilde{\beta}$ *and* $\tilde{\beta} \cdot \mathtt{()} = \tilde{\beta}$. *The last step gives us the result* $\gamma = \mathtt{a} \mid \mathtt{@} \cdot \mathtt{b?} \cdot \mathtt{@}$, *which is a regular expression in* `brics` *syntax that defines the same language as* α *in* `ECMA-262` *syntax.*

3 Evaluation and Discussion

JSON Schema patterns do not only occur with a clause `"pattern": regExp` but also as a pattern definition of a property name as mentioned in Section 2. Occurrences of both types are considered in the evaluation together. The numbers are based on a dataset of schemas found in open-source projects on GitHub [Ba21]. This corpus consists of 82,094 files. 17,747 of these files contain at least one pattern. This is 21.62%, so more than one fifth. Hence the goal to translate each JSON Schema pattern in an automaton-compatible syntax is relevant. After collecting all these patterns and eliminating duplicates we obtained 3,232 unique patterns. Our reproduction package is available at `https://doi.org/10.5281/zenodo.7586341`.

The implementation handles non- and named-capturing groups as capturing groups. In Table 2 we consider the numbers for syntactically invalid patterns and patterns for which we do not support a conversion. The former ones are 0.4% of the unique patterns and are mostly due to unescaped slashes, which are not allowed in `ECMA-262`. The latter ones are patterns containing word boundaries or lookaround, that is lookahead or lookbehind. These kind of patterns occurs in 2.44% of the unique patterns. These are the only possible cases for patterns which the algorithm cannot convert to `brics` syntax. For all the other ones, which is over 97% of the unique patterns, the procedure is successful.

In Table 3 we consider numbers for these `brics`-manageable patterns, where over 84% of them contain at least one anchor symbol, i.e. `^` or `$`. However, in nearly all of these patterns the anchors are not inside the regular expression, which means that `^` resp. `$` stands at the beginning resp. end of the expression. Only 0.67% of the `brics`-manageable patterns have anchors inside. If a pattern contains a non-nullable part before a starting or after an ending marker this pattern is unsatisfiable, i.e. it accepts the empty language. Fortunately, such patterns are scarce. Also nullable patterns, that are patterns which match the empty word (see Definition 2.1), are rare, they amount to less than 6% of the unique patterns.

As we can see in the experimental results, there are less than 3% of the unique patterns for which our approach fails. These are due to invalid patterns, word boundaries and lookaround. However, these kinds of patterns are also not supported by the former approach in [Ha21].

Tab. 2: Patterns extracted from the corpus.

	Total	%
Unique patterns	3,232	100.00
Invalid patterns	13	0.40
Not supported patterns	79	2.44
Manageable in `brics`	3,140	97.15

Tab. 3: Patterns manageable in `brics`.

	Total	%
Patterns with anchors	2,648	84.33
Anchors inside	21	0.67
Patterns without anchors	492	15.67
Nullable patterns (Def. 2.1)	183	5.83

4 Related Work

Our implementation is integrated in a tool which can generate witnesses for JSON Schema documents [At22]. This tool can be used to check schema containment. An earlier approach to containment checking (not based on witness generation) is presented in [Ha21] . It also relies on an external automaton library, the Python `greenery` library [4], which also uses a non-anchored syntax. Habib et al. unanchor the patterns from the schemas and unescape the anchor symbols before using `greenery` especially for computing intersections of two regular expressions. These steps are only executed on the string representation of the regular expression, without parsing its structure. Thus, there are instances when the approach by Habib et al. fails to preserve pattern semantics, e.g. for patterns containing anchors inside and not at its beginning or end – different from our well-principled approach.

5 Conclusion

We have successfully integrated our syntax conversion in a tool for JSON Schema witness generation. The experiments in [At22] reveal that the generation of automata from complex patterns in JSON Schema can cause severe performance problems. This motivates a range of follow-up research, for instance, caching of reoccurring patterns, or a lazy computation of automata for the purpose of witness generation.

Acknowledgments. This article describes the results of a bachelor thesis project at the *University of Passau*, supervised by Stefanie Scherzinger. I thank Dominik Freydenberger for suggesting the conversion algorithm. I further thank the authors of [At22], specifically Lyes Attouche, Mohamed Amine-Baazizi, Dario Colazzo, Giorgio Ghelli, and Dario Colazzo for guidance and support. This contribution was partly funded by *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) grant #385808805.

[4] `https://github.com/qntm/greenery`

References

[At22] Attouche, L.; Baazizi, M.-a.; Colazzo, D.; Ghelli, G.; Sartiani, C.; Scherzinger, S.: Witness Generation for JSON Schema. Proc. VLDB Endow. 15/13, pp. 4002–4014, 2022, URL: https://www.vldb.org/pvldb/vol15/p4002-sartiani.pdf.

[Ba21] Baazizi, M.-A.; Ghelli, G.; Colazzo, D.; Sartiani, C.; Scherzinger, S.: sdbs-uni-p/json-schema-corpus: JSON Schema Corpus Release 07/2021, using the updated data from https://github.com/sdbs-uni-p/json-schema-corpus, commit hash #79f808b, July 2021, URL: https://doi.org/10.5281/zenodo.5141199.

[FS19] Freydenberger, D. D.; Schmid, M. L.: Deterministic regular expressions with back-references. In: Journal of Computer and System Sciences. 105, pp. 1–39, 2019, URL: https://doi.org/10.1016/j.jcss.2019.04.001.

[Ha21] Habib, A.; Shinnar, A.; Hirzel, M.; Pradel, M.: Finding Data Compatibility Bugs with JSON Subschema Checking. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2021, pp. 620–632, 2021, URL: https://doi.org/10.1145/3460319.3464796.

[HU79] Hopcroft, J. E.; Ullman, J. D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company, 1979.

[Mø17] Møller, A.: dk.brics.automaton – Finite-State Automata and Regular Expressions for Java, version 1.12-1, 2017, URL: http://www.brics.dk/automaton/.

[Pe16] Pezoa, F.; Reutter, J. L.; Suarez, F.; Ugarte, M.; Vrgoč, D.: Foundations of JSON Schema. In: Proceedings of the 25th International Conference on World Wide Web. WWW '16, pp. 263–273, 2016, URL: https://doi.org/10.1145/2872427.2883029.